datajudge

Release 1.0

QuantCo Inc.

Apr 03, 2024

CONTENTS

`datajudge` allows for assessing whether data from database complies with reference information.

While meant to be as agnostic to concrete database management systems as possible, `datajudge` currently explicitly supports:

- Postgres
- MSSQL
- Snowflake

CHAPTER

ONE

CONTENTS

1.1 Installation

To install, execute

```
pip install datajudge
```

or from a conda environment

```
conda install datajudge -c conda-forge
```

1.2 Getting Started

1.2.1 Glossary

- A `DataSource` represents a way to retrieve data from database. Typically, this corresponds to a table in the database. Yet, it could also be a more elaborate object. See the section on 'Alternative `DataSource` s' for more detail.
- A `Constraint` captures a concrete expectation between either two `DataSource` s or a single `DataSource` and a reference value.
- A `Requirement` captures all `Constraint` s between two given `DataSource` s or all `Constraint` s within a single `DataSource`. If a `Requirement` refers links to two `DataSource` s, it is a `BetweenRequirement`. If a `Requirement` merely refers to a single `DataSource`, it is a `WithinRequirement`.
- Conceptually, a 'specification' captures all `Requirement` s against a database. In practice that means it is usually a separate python file which:
  - gathers all relevant `Requirement` s
  - turns these `Requirement` s' `Constraint` s into individual tests
  - can be 'tested' by pytest

1.2.2 Creating a specification

In order to get going, you might want to use the following snippet in a new python file. This file will represent a specification.

```
import pytest
import sqlalchemy as sa
from datajudge.pytest_integration import collect_data_tests


@pytest.fixture(scope="module")
def datajudge_engine():
    # TODO: Adapt connection string to database at hand.
    return sa.create_engine("your_connection_string")


# TODO: Insert Requirement objects to list.
requirements = []

test_constraints = collect_data_tests(requirements)
```

This file will eventually lead as an input to pytest. More on that in the section 'Testing a specification'.

In case you haven't worked with sqlalchemy engines before, you might need to install drivers to connect to your database. You might want to install snowflake-sqlalchemy when using Snowflake, pyscopg when using Postgres and platform-specific drivers (Windows, Linux, macOS) when using MSSQL.

1.2.3 Specifying Constraints

In order to discover possible `Constraint` s, please investigate the `_add_*_constraint` methods for *`BetweenRequirement`* and *`WithinRequirement`* respectively.

These methods are meant to be mostly self-documenting through the usage of expressive parameters.

Note that most `Constraint` s will allow for at least one `Condition`. A `Condition` can be thought of as a conditional event in probability theory or a filter/clause in a database query. Please consult the doc string of `Condition` for greater detail. For examples, please see `tests/unit/test_condition.py`.

Many `Constraint` s have optional `columns` parameters. If no argument is given, all available columns will be used.

1.2.4 Defining limitations of change

`BetweenRequirement` s allow for `Constraint` s expressing the limitation of a loss or gain. For example, the `NRowsMinGain` `Constraint` expresses by how much the number of rows must at least grow from the first `DataSource` to the second. In the example of `NRowsMinGain` , this growth limitation is expressed relative to the number of rows of the first `DataSource`.

Generally, such relative limitations can be defined in two ways:

- manually, based on domain knowledge (e.g. 'at least 5% growth')
- automatically, based on date ranges

The former would translate to

```
#rows_table_2 > (1 + min_relative_gain) * #rows_table_1
```

while the latter would translate to

```
date_growth := (max_date_table_2 - min_date_table_2) / (max_date_table_1 - min_date_
↪table_1)
#rows_table_2 > (1 + date_growth) * #rows_table_1
```

In the latter case a date column must be passed during the instantiation of the `BetweenRequirement`. Moreover, the `date_range_*` must be passed in the respective `add_*_constraint` method. When using date ranges as an indicator of change, the `constant_max_*` argument can safely be ignored. Additionally, an additional buffer to the date growth can be added with help of the `date_range_gain_deviation` parameter:

```
date_growth := (max_date_table_2 - min_date_table_2) / (max_date_table_1 - min_date_
↪table_1)
#rows_table_2 > (1 + date_growth + date_range_gain_deviation) + * #rows_table_1
```

This example revolving around `NRowsMinGain` generalizes to many `Constraint` s concerned with growth, gain, loss or shrinkage limitations.

1.2.5 Testing a specification

In order to test whether the `Constraint` s expressed in a specification hold true, you can simply run

```
pytest your_specification.py
```

This will produce results directly in your terminal. If you prefer to additionally generate a report, you can run

```
pytest your_specification.py --html=your_report.html
```

As the testing relies on pytest, all of *pytest*'s features can be used. More on this in the article on *testing*.

1.2.6 Test information

When calling a `Constraint`'s `test` method, a `TestResult` is returned. The latter comes with a `logging_message` field. This field comprises information about the test failure, the constraint at hand as well as the underlying database queries.

Depending on the use case at hand, it might make sense to rely on this information for logging or data investigation purposes. Again, more on this in the article on *testing*.

1.2.7 Assertion Message Styling

Constraints can use styling to increase the readability of their assertion messages. The styling can be set independently of the platform and converted to e.g. ANSI color codes for command line output or CSS color tags for HTML reports. The styling tags describe use cases and not concrete colors, so formatters can use arbitrary color palettes, and these are not fixed by the constraint.

The following table lists all the supported codes, along with their descriptions and examples of how they can be used:

Table 1: Supported styling codes

Code	Description	Example
numMatch	Indicates the part of a number that matches the expected value.	*[numMatch]3.141[/numMatch]*
numDiff	Indicates the part of a number that differs.	*[numDiff]6[/numDiff]*

1.2.8 Alternative DataSources

A `Requirement` is instantiated with either one or two fixed `DataSource` s.

While the most typical example of a `DataSource` would be a table in a database, `datajudge` allows for other `DataSource` s as well. These are often derived from primitive tables of a database.

Table 2: DataSources

`DataSource`	explanation	*`WithinRequirement`* constructor	*`BetweenRequirement`* constructor
`TableDataSource`	represents a table in a database	`from_table()`	`from_tables()`
`ExpressionDataS`	represents the result of a `sqlalchemy` expression	`from_expression()`	`from_expressions()`
`RawQueryDataSou`	represents the result of a sql query expressed via a string	`from_raw_query()`	`from_raw_queries()`

Typically, a user does not need to instantiate a corresponding `DataSource` themselves. Rather, this is taken care of by using the appropriate constructor for `WithinRequirement` or `BetweenRequirement`.

Note that in principle, several tables can be combined to make up for a single `DataSource`. Yet, most of the time when trying to compare two tables, it is more convenient to create a `BetweenRequirement` and use the `from_tables` constructor.

1.2.9 Column capitalization

Different database management systems handle the capitalization of entities, such as column names, differently. For the time being:

- Mssql: `datajudge` expects column name capitalization as is seen in database, either lowercase or uppercase.
- Postgres: `datajudge` expects lowercase column names.
- Snowflake: `datajudge` will lowercase independently of the capitalization provided.

The Snowflake behavior is due to an upstream bug in snowflake-sqlalchemy.

This behavior is subject to change.

1.3 Testing

While `datajudge` allows to express expectations via specifications, `Requirement` s and `Constraint` s, the execution of tests is delegated to pytest. As a consequence, one may use any functionalities that pytest has to offer. Here, we want to illustrate some of these advanced functionalities that might turn out useful.

Yet, it should be noted that for most intents and purposes, using `datajudge` 's helper function *`collect_data_tests()`* is a good starting point. It should work out of the box and hides some complexity. For exemplary applications see, the *companies example* or the *twitch example*.

Throughout this article we will not rely on `collect_data_tests`. Instead we will more explicitly create a mechanism turning a List of `Requirement` objects into something that can be tested by pytest manually. Importantly, we want every `Constraint` of every `Requirement` to be tested independently of each other. For instance, we would not like one failing test to halt all others.

Many of these approaches rely on adapting pytest's `conftest.py`. If you are not familiar with this concept, you might want to read up on it here.

1.3.1 Subselection

Most often one might want to run all tests defined by a specification.

Yet, for example after believing to have fixed a data problem, one might simply want to test whether a single test, which had previously been failing, succeeds at last.

Another example for when one would like to test a subset of tests is if the data at hand is not available in its entirety. Rather, it could be that one would like to run a subset of the test suite against a subsample of the typical dataset.

In this section, we present two approaches to do a subselection of tests.

Ex-post: subselecting generated tests

Instead of merely running `$ pytest specification.py` one may add pytests's `-k` flag and specify the `Constraint` (s) one cares about.

Importantly, every `Constraint` object can be identified via a name. If one wants to figure out how this string is built, please refer to the implementation of `get_description()`. Otherwise, one could also just run all of the tests once and investigate the resulting test report to find the relevant names.

When only caring about the `UniquesEquality` constraint in our *twitch example*. one might for instance use the following prefix the filter for it:

```
$ pytest twitch_specification.py -k "UniquesEquality::public.twitch_v1"
```

Ex-ante: Defining categories of tests

Another option to subselect a certain set of tests is by use of pytest markers. The following is one way of using markers in conjunction with `datajudge`.

In this particular illustration we'll allow for two markers:

- `basic`: indicating that only truly fundamental tests should be run
- `all`: indicating that any available test should be run

For that matter we'll add a bit of pytest magic to the respective `conftest.py`.

Listing 1: `conftest.py`

```
def pytest_generate_tests(metafunc):
    if "basic_constraint" in metafunc.fixturenames:
        metafunc.parametrize(
            "basic_constraint",
            # Find these functions in specification.py.
            metafunc.module.get_basic_constraints(),
            ids=metafunc.module.idfn,
        )
    if "constraint" in metafunc.fixturenames:
        metafunc.parametrize(
            "constraint",
            # Find these functions in specification.py.
            metafunc.module.get_all_constraints(),
            ids=metafunc.module.idfn,
        )
```

Moreover, we'll have to register these markers in pytest's `pytest.ini` file. You can read more about these files here.

Listing 2: `pytest.ini`

```
[pytest]
addopts = --strict-markers
markers = basic: basic specification
          all: entire specification
```

Once that is taken care of, one can adapt one's specification as follows:

Listing 3: `specification.py`

```
def get_basic_requirements() -> List[Requirement]:
    # Create relevant Requirement objects and respective Constraints.
    # ...

    return requirements

def get_advanced_requirements() -> List[Requirement]:
    # Create relevant Requirement objects and respective Constraints.
    # ...

    return requirements

def get_basic_constraints() -> List[Constraint]:
    return [constraint for requirement in get_basic_requirements() for constraint in␣
↪requirement]

def get_all_constraints() -> List[Constraint]:
    all_requirements = get_basic_requirements() + get_advanced_requirements()
    return [constraint for requirement in all_requirements for constraint in requirement]

# Function used in conftest.py.
# Given a constraint, returns an identifier used to refer to it as a test.
def idfn(constraint):
    return constraint.get_description()

@pytest.mark.basic
def test_basic_constraint(basic_constraint: Constraint, datajudge_engine):
    test_result = basic_constraint.test(datajudge_engine)
    assert test_result.outcome, test_result.failure_message

@pytest.mark.all
def test_all_constraint(constraint: Constraint, datajudge_engine):
    test_result = constraint.test(datajudge_engine)
    assert test_result.outcome, test_result.failure_message
```

Once these changes are taken care of, one may run

```
$ pytest specification.py -m basic
```

to only test the basic `Requirement` s or

```
$ pytest specification.py -m all
```

to test all `Requirement` s.

1.3.2 Using parameters in a specification

A given specification might rely on identifiers such as database names or table names. Moreover it might be that, e.g. when iterating from one version of the data to another, these names change.

In other words, it could be that the logic should remain unchanged while pointers to data might change. Therefore, one might just as well consider those pointers or identifiers as parameters of the specification.

For the sake of concreteness, we will assume here that we wish frame two identifiers as parameters:

- `new_db`: the name of the 'new database'
- `old_db`: the name of the 'old database'

In light of that we will again adapt pytest's `conftest.py`:

Listing 4: `conftest.py`

```
def pytest_addoption(parser):
    parser.addoption("--new_db", action="store", help="name of the new database")
    parser.addoption("--old_db", action="store", help="name of the old database")


def pytest_generate_tests(metafunc):
    params = {
        "db_name_new": metafunc.config.option.new_db,
        "db_name_old": metafunc.config.option.old_db,
    }
    metafunc.parametrize(
        "constraint",
        metafunc.module.get_constraints(params),
        ids=metafunc.module.idfn,
    )
```

Now, we can make the creation of our `Requirement` s and `Constraint` s dependent on these parameters:

Listing 5: `specification.py`

```
def get_requirements(params):
    between_requirement = BetweenRequirement.from_tables(
        db_name1=params["old_db"],
        db_name2=params["new_db"],
        # ...
    )
    # ...
    return requirements

def get_constraints(params):
    return [
        constraint for requirement in get_requirements(params) for constraint in␣
↪requirement
```

(continues on next page)

(continued from previous page)

```
    ]

def idfn(constraint):
    return constraint.get_description()

def test_constraint(constraint, datajudge_engine):
    test_result = constraint.test(datajudge_engine)
    assert test_result.outcome, test_result.failure_message
```

Once the specification is defined to be dependent on such parameters, they can simply be passed via CLI:

```
$ pytest specification.py --new_db=db_v1 --old_db=db_v2
```

1.3.3 Html reports

By default, running `pytest` tests will output test results to one's respective shell. Alternatively, one might want to generate an html report summarizing and expanding on all test results. This can be advantageous for

- Sharing test results with colleagues
- Archiving and tracking test results over time
- Make underlying sql queries conveniently accessible

Concretely, such an html report can be generated by pytest-html. Once installed, using it is as simple as appending `--html=myreport.html` to the pytest call.

In our twitch example, this generates this html report.

1.3.4 Retrieving queries

Usually we not only care about knowing whether there is a problem with the data at hand and what it is. Rather, we would also like to fix it as fast and conveniently as possible.

For that matter, `datajudge` makes the queries it uses to assert testing predicates available via the `datajudge.constraints.base.TestResult` class. Hence, if a test is failing, the user can jumpstart the investigation of the problem by reusing and potentially adapting the underlying queries.

Instead of simply running `assert constraint.test(engine).outcome`, one may add the `TestResult` 's `logging_message` to e.g. a `logger` or add it to pytest `extra`:

```
from pytest_html import extras

def test_constraint(constraint: Constraint, engine, extra):
  test_result = constraint.test(engine)
  message = test_result.logging_message

  if not test_result.outcome:
    # Send to logger.
    logger.info(message)
    # Add to html report.
    extra.append(
      extras.extra(
        content=message,
```

(continues on next page)

(continued from previous page)

```
        format_type="text",
        name="failing_query",
        mime_type="text/plain",
        extension="sql",
      )
    )

 assert test_result.outcome
```

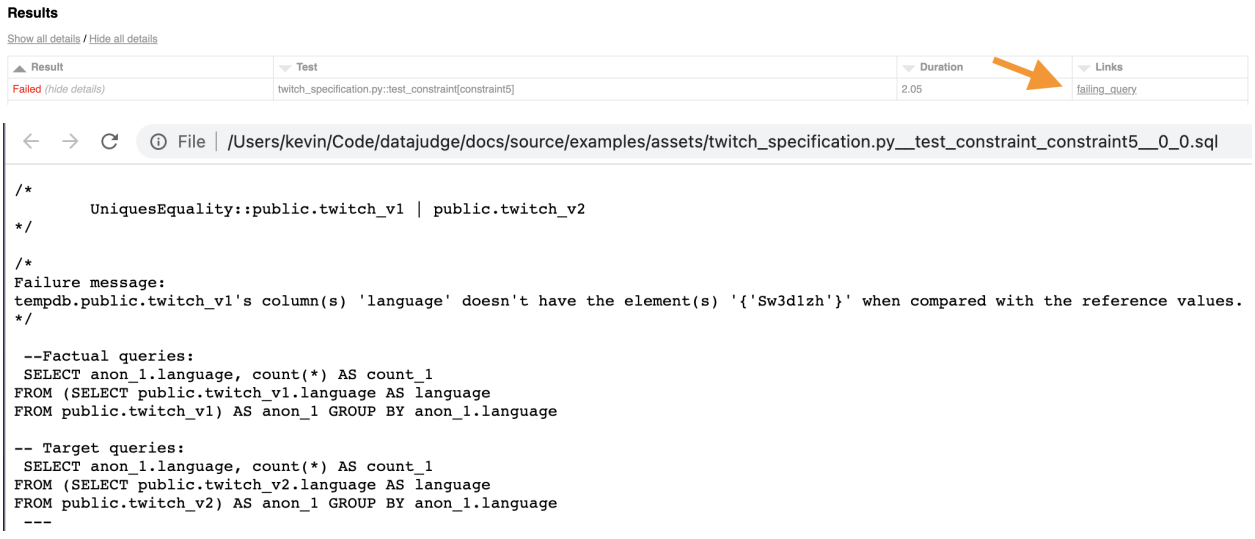
Such a `logging_message` - with ready to execute sql queries - can look as follows:

```
/*
Failure message:
tempdb.public.twitch_v1's column(s) 'language' doesn't have the
element(s) '{'Sw3d1zh'}' when compared with the reference values.
*/

 --Factual queries:
 SELECT anon_1.language, count(*) AS count_1
FROM (SELECT public.twitch_v1.language AS language
FROM public.twitch_v1) AS anon_1 GROUP BY anon_1.language

-- Target queries:
 SELECT anon_1.language, count(*) AS count_1
FROM (SELECT public.twitch_v2.language AS language
FROM public.twitch_v2) AS anon_1 GROUP BY anon_1.language
```

If using a mechanism - as previously outlined - to forward these messages to an html report, this can look as follows:



1.4 Motivation

Ensuring data quality is of great importance for many use cases. `datajudge` seeks to make this convenient.

`datajudge` allows for the expression of expectations held against data stored in databases. In particular, it allows for comparing different `DataSource` s. Yet, it also comes with functionalities to compare data from a single `DataSource` to fixed reference values derived from explicit domain knowledge.

Not trying to reinvent the wheel, `datajudge` relies on `pytest` to execute the data expectations.

1.4.1 Comparisons between DataSources

The data generating process can be obscure for a variety of reasons. In such scenarios one might ask the questions of

- Has the data 'changed' over time?
- Was the transformation of the data successful?

In both cases one might want to compare different data – either from different points in time or from different transformation steps – to each other.

1.4.2 Why not Great Expectations?

The major selling point is to be able to conveniently express expectations **between** different `DataSource` s. Great Expectations, in contrast, focuses on expectations against a single `DataSource`.

Moreover, some users have pointed out the following advantages:

- lots of 'query writing' is taken care of by having tailored `Constraint` s
- easier and faster onboarding
- assertion messages with counterexamples and other context information, speeding up the data debugging process

1.5 Example: Company data

To get started, we will create a sample database using sqlite that contains a list of companies.

The table "companies_archive" contains three entries:

Table 3: companies_archive

id	name	num_employees
1	QuantCo	90
2	Google	140,000
3	BMW	110,000

While "companies" contains an additional entry:

Table 4: companies

id	name	num_employees
1	QuantCo	100
2	Google	150,000
3	BMW	120,000
4	Apple	145,000

```
import sqlalchemy as sa

eng = sa.create_engine('sqlite:///example.db')

with eng.connect() as con:
    con.execute("CREATE TABLE companies (id INTEGER PRIMARY KEY, name TEXT, num_
↪employees INTEGER)")
    con.execute("INSERT INTO companies (name, num_employees) VALUES ('QuantCo', 100), (
↪'Google', 150000), ('BMW', 120000), ('Apple', 145000)")
    con.execute("CREATE TABLE companies_archive (id INTEGER PRIMARY KEY, name TEXT, num_
↪employees INTEGER)")
    con.execute("INSERT INTO companies_archive (name, num_employees) VALUES ('QuantCo',␣
↪90), ('Google', 140000), ('BMW', 110000)")
```

As an example, we will run 4 tests on this table:

1. Does the table “companies” contain a column named “name”?
2. Does the table “companies” contain at least 1 entry with the name “QuantCo”?
3. Does the column “num_employees” of the “companies” table have all positive values?
4. Does the column “name” of the table “companies” contain at least all the values of the corresponding column in “companies_archive”?

```
import pytest
import sqlalchemy as sa

from datajudge import (
    Condition,
    WithinRequirement,
    BetweenRequirement,
)
from datajudge.pytest_integration import collect_data_tests


# We create a Requirement, within a table. This object will contain
# all the constraints we want to test on the specified table.
# To test another table or test the same table against another table,
# we would create another Requirement object.
companies_req = WithinRequirement.from_table(
    db_name="example", schema_name=None, table_name="companies"
)

# Constraint 1: Does the table "companies" contain a column named "name"?
companies_req.add_column_existence_constraint(columns=["name"])
```

(continues on next page)

(continued from previous page)

```
# Constraint 2: Does the table "companies" contain at least 1 entry with the name
↪"QuantCo"?
condition = Condition(raw_string="name = 'QuantCo'")
companies_req.add_n_rows_min_constraint(n_rows_min=1, condition=condition)

# Constraint 3: Does the column "num_employees" of the "companies" table have all
# positive values?
companies_req.add_numeric_min_constraint(column="num_employees", min_value=1)

# We create a new Requirement, this time between different tables.
# Concretely, we intent to test constraints between the table "companies"
# and the table "companies_archive".
companies_between_req = BetweenRequirement.from_tables(
    db_name1="example",
    schema_name1=None,
    table_name1="companies",
    db_name2="example",
    schema_name2=None,
    table_name2="companies_archive",
)

# Constraint 4: Does the column "name" of the table "companies" contain at least all
# the values of the corresponding column in "companies_archive"?
companies_between_req.add_row_superset_constraint(
    columns1=['name'], columns2=['name'], constant_max_missing_fraction=0
)

# collect_data_tests expects a pytest fixture with the name
# "datajudge_engine" that is a SQLAlchemy engine

@pytest.fixture()
def datajudge_engine():
    return sa.create_engine("sqlite:///example.db")

# We gather our distinct Requirements in a list.
requirements = [companies_req, companies_between_req]

# "collect_data_tests" takes all requirements and turns their respective
# Constraints into individual tests. pytest will be able to pick
# up these tests.
test_constraint = collect_data_tests(requirements)
```

Saving this file as `specification.py` and running `$ pytest specification.py` will verify that all constraints are satisfied. The output you see in the terminal should be similar to this:

```
================================== test session starts␣
↪==================================
...
collected 4 items

specification.py::test_constraint[ColumnExistence::companies] PASSED                [ 25
```

(continues on next page)

(continued from previous page)

```
↪%]
specification.py::test_constraint[NRowsMin::companies] PASSED                    [ 50
↪%]
specification.py::test_constraint[NumericMin::companies] PASSED                  [ 75
↪%]
specification.py::test_constraint[RowSuperset::companies|companies_archive] PASSED  [100
↪%]

==================================== 4 passed in 0.31s␣
↪====================================
```

You can also use a formatted html report using the `--html=report.html` flag.

1.6 Example: Dumps of Twitch data

This example is based on data capturing statistics and properties of popular Twitch channels. The setup is such that we have two data sets 'of the same kind' but from different points in time.

In other words, a 'version' of the data set represents a temporal notion. For example, version 1 might stem from end of March and version 2 from end of April. Moreover, we will assume that the first, version 1, has been vetted and approved with the help of manual investigation and domain knowledge. The second data set, version 2, has just been made available. We would like to use it but can't be sure of its validity just yet. As a consequence we would like to assess the quality of the data in version 2.

In order to have a database Postgres instance to begin with, it might be useful to use our script, spinning up a dockerized Postgres database:

```
$ ./start_postgres.sh
```

The original data set can be found on kaggle. For the sake of this tutorial, we slightly process it and provide two versions of it. One can either recreate this by executing this processing script oneself on the original data or download our processed files (version 1 and version 2) right away.

Once both version of the data exist, they can be uploaded to the tabase. We provide an uploading script creating and populating one table per version of the data in a Postgres database. It resembles the following:

```
address = os.environ.get("DB_ADDR", "localhost")
connection_string = f"postgresql://datajudge:datajudge@{address}:5432/datajudge"
engine = sa.create_engine(connection_string)
df_v2.to_sql("twitch_v2", engine, schema="public", if_exists="replace")
df_v1.to_sql("twitch_v1", engine, schema="public", if_exists="replace")
```

Once the tables are stored in a database, we can actually write a `datajudge` specification against them. But first, we'll have a look at what the data roughly looks like by investigating a random sample of four rows:

Table 5: A sample of the data

chan-nel	watch_t	stream_t	peak_vie	aver-age_viewe	fol-low-ers	follow-ers_gained	views_ga	part-nered	ma-ture	lan-guage
xQ-cOW	6196161	215250	222720	27716	324629	1734810	93036735	True	False	En-glish
sum-mit1g	6091677	211845	310998	25610	531016	1374810	89705964	True	False	En-glish
Gaules	5644590	515280	387315	10976	176763	1023779	10261160	True	True	Por-tuguese
ESL_CS	3970318	517740	300575	7714	394485	703986	10654694	True	False	En-glish

Note that we expect both version 1 and version 2 to follow this structure. Due to them being assembled at different points in time, merely their rows shows differ.

Now let's write an actual specification, expressing our expectations against the data. First, we need to make sure a connection to the database can be established at test execution time. How this is done exactly depends on how you set up your database. When using our default setup with running, this would look as follows:

```
import os
import pytest
import sqlalchemy as sa


@pytest.fixture(scope="module")
def datajudge_engine():
    address = os.environ.get("DB_ADDR", "localhost")
    connection_string = f"postgresql://datajudge:datajudge@{address}:5432/datajudge"
    return sa.create_engine(connection_string)
```

Once a way to connect to the database is defined, we want to declare our data sources and express expectations against them. In this example, we have two tables in the same database - one table per version of the Twitch data.

Yet, let's start with a straightforward example only using version 2. We want to use our domain knowledge that constrains the values of the `language` column only to contain letters and have a length strictly larger than 0.

```
from datajudge import WithinRequirement


# Postgres' default database.
db_name = "tempdb"
# Postgres' default schema.
schema_name = "public"

within_requirement = WithinRequirement.from_table(
    table_name="twitch_v2",
    schema_name=schema_name,
    db_name=db_name,
)
within_requirement.add_varchar_regex_constraint(
    column="language",
```

(continues on next page)

(continued from previous page)

```
    regex="^[a-zA-Z]+$",
)
```

Done! Now onto comparisons between the table representing the approved version 1 of the data and the to be assessed version 2 of the data.

```
from datajudge import BetweenRequirement, Condition

between_requirement_version = BetweenRequirement.from_tables(
    db_name1=db_name,
    db_name2=db_name,
    schema_name1=schema_name,
    schema_name2=schema_name,
    table_name1="twitch_v1",
    table_name2="twitch_v2",
)
between_requirement_version.add_column_subset_constraint()
between_requirement_version.add_column_superset_constraint()
columns = ["channel", "partnered", "mature"]
between_requirement_version.add_row_subset_constraint(
    columns1=columns, columns2=columns, constant_max_missing_fraction=0
)
between_requirement_version.add_row_matching_equality_constraint(
    matching_columns1=["channel"],
    matching_columns2=["channel"],
    comparison_columns1=["language"],
    comparison_columns2=["language"],
    max_missing_fraction=0,
)

between_requirement_version.add_ks_2sample_constraint(
    column1="average_viewers",
    column2="average_viewers",
    significance_level=0.05,
)
between_requirement_version.add_uniques_equality_constraint(
    columns1=["language"],
    columns2=["language"],
)
```

Now having compared the 'same kind of data' between version 1 and version 2, we may as well compare 'different kind of data' within version 2, as a means of a sanity check. This sanity check consists of checking whether the mean `average_viewer` value of mature channels should deviate at most 10% from the overall mean.

```
between_requirement_columns = BetweenRequirement.from_tables(
    db_name1=db_name,
    db_name2=db_name,
    schema_name1=schema_name,
    schema_name2=schema_name,
    table_name1="twitch_v2",
    table_name2="twitch_v2",
)
```

(continues on next page)

(continued from previous page)

```
between_requirement_columns.add_numeric_mean_constraint(
    column1="average_viewers",
    column2="average_viewers",
    condition1=None,
    condition2=Condition(raw_string="mature IS TRUE"),
    max_absolute_deviation=0.1,
)
```

Lastly, we need to collect all of our requirements in a list and make sure `pytest` can find them by calling `collect_data_tests`.

```
from datajudge.pytest_integration import collect_data_tests
requirements = [
    within_requirement,
    between_requirement_version,
    between_requirement_columns,
]
test_func = collect_data_tests(requirements)
```

If we then test these expectations against the data by running `$ pytest specification.py` – where `specification.py` contains all of the code outlined before (you can find it here) – we see that the new version of the data is not quite on par with what we'd expect:

```
$ pytest twitch_specification.py
================================== test session starts␣
↪===================================
platform darwin -- Python 3.10.5, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/kevin/Code/datajudge/docs/source/examples
plugins: html-3.1.1, cov-3.0.0, metadata-2.0.2
collected 8 items

twitch_specification.py F.....FF                                          [100%]

======================================== FAILURES␣
↪=========================================
_____________________ test_func[VarCharRegex::tempdb.public.twitch_v2] _____________________
↪_

constraint = <datajudge.constraints.varchar.VarCharRegex object at 0x10855da20>
datajudge_engine = Engine(postgresql://datajudge:***@localhost:5432/datajudge)

    @pytest.mark.parametrize(
        "constraint", all_constraints, ids=Constraint.get_description
    )
    def test_constraint(constraint, datajudge_engine):
        test_result = constraint.test(datajudge_engine)
>       assert test_result.outcome, test_result.failure_message
E       AssertionError: tempdb.public.twitch_v2's column(s) 'language' breaks regex
        '^[a-zA-Z]+$' in 0.045454545454545456 > 0.0 of the cases. In absolute terms, 1
        of the 22 samples violated the regex. Some counterexamples consist of the
        following: ['Sw3d1zh'].
```

(continues on next page)

(continued from previous page)

```
../../../src/datajudge/pytest_integration.py:25: AssertionError
_____________ test_func[UniquesEquality::public.twitch_v1 | public.twitch_v2] _____________
↪_

constraint = <datajudge.constraints.uniques.UniquesEquality object at 0x10855d270>
datajudge_engine = Engine(postgresql://datajudge:***@localhost:5432/datajudge)

    @pytest.mark.parametrize(
        "constraint", all_constraints, ids=Constraint.get_description
    )
    def test_constraint(constraint, datajudge_engine):
        test_result = constraint.test(datajudge_engine)
>       assert test_result.outcome, test_result.failure_message
E       AssertionError: tempdb.public.twitch_v1's column(s) 'language' doesn't have
        the element(s) '{'Sw3d1zh'}' when compared with the reference values.

../../../src/datajudge/pytest_integration.py:25: AssertionError
_____________ test_func[NumericMean::public.twitch_v2 | public.twitch_v2] _____________
↪_

constraint = <datajudge.constraints.numeric.NumericMean object at 0x1084e1810>
datajudge_engine = Engine(postgresql://datajudge:***@localhost:5432/datajudge)

    @pytest.mark.parametrize(
        "constraint", all_constraints, ids=Constraint.get_description
    )
    def test_constraint(constraint, datajudge_engine):
        test_result = constraint.test(datajudge_engine)
>       assert test_result.outcome, test_result.failure_message
E       AssertionError: tempdb.public.twitch_v2's column(s) 'average_viewers' has
        mean 4734.9780000000000000, deviating more than 0.1 from
        tempdb.public.twitch_v2's column(s) 'average_viewers''s
        3599.9826086956521739. Condition on second table: WHERE mature IS TRUE

../../../src/datajudge/pytest_integration.py:25: AssertionError
================================== short test summary info␣
↪==================================
FAILED twitch_specification.py::test_func[VarCharRegex::tempdb.public.twitch_v2] - Asse..
↪.
FAILED twitch_specification.py::test_func[UniquesEquality::public.twitch_v1 | public.
↪twitch_v2]
FAILED twitch_specification.py::test_func[NumericMean::public.twitch_v2 | public.twitch_
↪v2]
============================== 3 failed, 5 passed in 1.52s␣
↪==================================
```

Alternatively, you can also look at these test results in this html report generated by pytest-html.

Hence we see that we might not want to blindly trust version 2 of the data as is. Rather, we might need to investigate what is wrong with the data, what this has been caused by and how to fix it.

Concretely, what exactly do we learn from the error messages?

- The column `language` now has a row with value `'Sw3d1zh'`. This break two of our constraints. The `VarCharRegex` constraint compared the columns' values to a regular expression. The `UniquesEquality` constraint expected the unique values of the `language` column to not have changed between version 1 and version 2.
- The mean value of `average_viewers` of `mature` channels is substantially - more than our 10% tolerance - lower than the global mean.

1.7 Example: Dates

This example concerns itself with expressing `Constraints` against data revolving around dates. While date `Constraints` between tables exist, we will only illustrate `Constraints` on a single table and reference values here. As a consequence, we will only use `WithinRequirement`, as opposed to `BetweenRequirement`.

Concretely, we will assume a table containing prices for a given product of id 1. Importantly, these prices are valid for a certain date range only. More precisely, we assume that the price for a product - identified via the `preduct_id` column - is indicated in the `price` column, the date from which it is valid - the date itself included - in `date_from` and the the until when it is valid - the date itself included - in the `date_to` column.

Such a table might look as follows:

Table 6: prices

product_id	price	date_from	date_to
1	13.99	22/01/01	22/01/10
1	14.5	22/01/11	22/01/17
1	13.37	22/01/16	22/01/31

Given this table, we would like to ensure - for the sake of illustrational purposes - that 6 constraints are satisfied:

1. All values from column `date_from` should be in January 2022.
2. All values from column `date_to` should be in January 2022.
3. The minimum value in column `date_from` should be the first of January 2022.
4. The maximum value in column `date_to` should be the 31st of January 2022.
5. There is no gap between `date_from` and `date_to`. In other words, every date of January has to be assigned to at least one row for a given product.
6. There is no overlap between `date_from` and `date_to`. In other words, every date of January has to be assigned to at most one row for a given product.

Assuming that such a table exists in database, we can write a specification against it.

```
import pytest
import sqlalchemy as sa

from datajudge import WithinRequirement
from datajudge.pytest_integration import collect_data_tests

# We create a Requirement, within a table. This object will contain
# all the constraints we want to test on the specified table.
# To test another table or test the same table against another table,
# we would create another Requirement object.
```

(continues on next page)

(continued from previous page)

```
prices_req = WithinRequirement.from_table(
    db_name="example", schema_name=None, table_name="prices"
)

# Constraint 1:
# All values from column date_from should be in January 2022.
prices_req.add_date_between_constraint(
    column="date_from",
    lower_bound="'20220101'",
    upper_bound="'20220131'",
    # We don't tolerate any violations of the constraint:
    min_fraction=1,
)

# Constraint 2:
# All values from column date_to should be in January 2022.
prices_req.add_date_between_constraint(
    column="date_to",
    lower_bound="'20220101'",
    upper_bound="'20220131'",
    # We don't tolerate any violations of the constraint:
    min_fraction=1,
)

# Constraint 3:
# The minimum value in column date_from should be the first of January 2022.

# Ensure that the minimum is smaller or equal the reference value min_value.
prices_req.add_date_min_constraint(column="date_from", min_value="'20220101'")
# Ensure that the minimum is greater or equal the reference value min_value.
prices_req.add_date_min_constraint(
    column="date_from",
    min_value="'20220101'",
    use_upper_bound_reference=True,
)

# Constraint 4:
# The maximum value in column date_to should be the 31st of January 2022.

# Ensure that the maximum is greater or equal the reference value max_value.
prices_req.add_date_max_constraint(column="date_to", max_value="'20220131'")
# Ensure that the maximum is smaller or equal the reference value max_value.
prices_req.add_date_max_constraint(
    column="date_to",
    max_value="'20220131'",
    use_upper_bound_reference=True,
)

# Constraint 5:
# There is no gap between date_from and date_to. In other words, every date
# of January has to be assigned to at least one row for a given product.
prices_req.add_date_no_gap_constraint(
```

(continues on next page)

(continued from previous page)

```
    start_column="date_from",
    end_column="date_to",
    # We don't want a gap of price date ranges for a given product.
    # For different products, we allow arbitrary date gaps.
    key_columns=["product_id"],
    # As indicated in prose, date_from and date_to are included in ranges.
    end_included=True,
    # Again, we don't expect any violations of our constraint.
    max_relative_violations=0,
)

# Constraint 6:
# There is no overlap between date_from and date_to. In other words, every
# of January has to be assigned to at most one row for a given product.
princes_req.add_date_no_overlap_constraint(
    start_column="date_from",
    end_column="date_to",
    # We want no overlap of price date ranges for a given product.
    # For different products, we allow arbitrary date overlaps.
    key_columns=["product_id"],
    # As indicated in prose, date_from and date_to are included in ranges.
    end_included=True,
    # Again, we don't expect any violations of our constraint.
    max_relative_violations=0,
)

@pytest.fixture()
def datajudge_engine():
    # TODO: Insert actual connection string
    return sa.create_engine("your_db://")

# We gather our single Requirement in a list.
requirements = [prices_req]

# "collect_data_tests" takes all requirements and turns their respective
# Constraints into individual tests. pytest will be able to pick
# up these tests.
test_constraint = collect_data_tests(requirements)
```

Pleas note that the `DateNoOverlap` and `DateNoGap` constraints also exist in a slightly different form: `DateNoOverlap2d` and `DateNoGap2d`. As the names suggest, these can operate in 'two date dimensions'.

For example, let's assume a table with four date columns, representing two ranges in distinct dimensions, respectively:

- `date_from`: Date from when a price is valid
- `date_to`: Date until when a price is valid
- `date_definition_from`: Date when a price definition was inserted
- `date_definition_to`: Date until when a price definition was used

Analogously to the unidimensional scenario illustrated here, one might care for certain constraints in two dimensions.

1.8 Example: Exploration

While datajudge seeks to tackle the use case of expressing and evaluating tests against data, its fairly generic inner workings allow for using it in a rather explorative workflow as well.

Let's first clarify terminology by exemplifying both scenarios. A person wishing to test data might ask the question

> Has the number of rows not grown too much from version 1 of the table to version 2 of the table?

whereas a person wishing to explore the data might ask the question

> By how much has the number of rows grown from version 1 to version 2 of the table?

Put differently, a test typically revolves around a binary outcome while an exploration usually doesn't.

In the following we will attempt to illustrate possible usages of datajudge for exploration by looking at three simple examples.

These examples rely on some insight about how most datajudge `Constraint` s work under the hood. Importantly, `Constraint` s typically come with

- a `retrieve` method: this method fetches relevant data from database, given a `DataReference`
- a `get_factual_value` method: this is typically a wrapper around `retrieve` for the first `DataReference` of the given `Requirement` / `Constraint`
- a `get_target_value` method: this is either a wrapper around `retrieve` for the second `DataReference` in the case of a `BetweenRequirement` or an echoing of the `Constraint` s key reference value in the case of a `WithinRequirement`

Moreover, as is the case when using datajudge for testing purposes, these approaches rely on a sqlalchemy engine. The latter is the gateway to the database at hand.

1.8.1 Example 1: Comparing numbers of rows

Assume we have two tables in the same database called `table1` and `table2`. Now we would like to compare their numbers of rows. Naturally, we would like to retrieve the respective numbers of rows before we can compare them. For this purpose we create a `BetweenTableRequirement` referring to both tables and add a `NRowsEquality` `Constraint` onto it.

```
import sqlalchemy as sa
from datajudge import BetweenRequirement

engine = sa.create_engine(your_connection_string)
req = BetweenRequirement.from_tables(
    db_name,
    schema_name,
    "table1",
    db_name,
    schema_name,
    "table2",
)
req.add_n_rows_equality_constraint()
n_rows1 = req[0].get_factual_value(engine)
n_rows2 = req[0].get_target_value(engine)
```

Note that here, we access the first (and only) `Constraint` that has been added to the `BetweenRequirement` by writing `req[0]`. `Requirements` are are sequences of `Constraint` s, after all.

Once the numbers of rows are retrieved, we can compare them as we wish. For instance, we could compute the absolute and relative growth (or loss) of numbers of rows from `table1` to `table2`:

```
absolute_change = abs(n_rows2 - n_rows1)
relative_change = (absolute_change) / n_rows1 if n_rows1 != 0 else None
```

Importantly, many datajudge staples, such as `Condition` s can be used, too. We shall see this in our next example.

1.8.2 Example 2: Investigating unique values

In this example we will suppose that there is a table called `table` consisting of several columns. Two of its columns are supposed to be called `col_int` and `col_varchar`. We are now interested in the unique values in these two columns combined. Put differently, we are wondering:

> Which unique pairs of values in `col_int` and `col_varchar` have we encountered?

To add to the mix, we will moreover only be interested in tuples in which `col_int` has a value of larger than 10.

As before, we will start off by creating a `Requirement`. Since we are only dealing with a single table this time, we will create a `WithinRequirement`.

```
import sqlalchemy as sa
from datajudge import WithinRequirement, Condition

engine = sa.create_engine(your_connection_string)

req = requirements.WithinRequirement.from_table(
    db_name,
    schema_name,
    "table",
)

condition = Condition(raw_string="col_int >= 10")

req.add_uniques_equality_constraint(
    columns=["col_int", "col_varchar"],
    uniques=[], # This is really just a placeholder.
    condition=condition,
)
uniques = req[0].get_factual_value(engine)
```

If one was to investigate this `uniques` variable further, one could, e.g. see the following:

```
([(10, 'hi10'), (11, 'hi11'), (12, 'hi12'), (13, 'hi13'), (14, 'hi14'), (15, 'hi15'),␣
↪(16, 'hi16'), (17, 'hi17'), (18, 'hi18'), (19, 'hi19')], [1, 100, 12, 1, 7, 8, 1, 1,␣
↪1337, 1])
```

This becomes easier to parse when inspecting the underlying `retrieve` method of the `UniquesEquality Constraint`: the first value of the tuple corresponds to the list of unique pairs in columns `col_int` and `col_varchar`. The second value of the tuple are the respective counts thereof.

Moreoever, one could manually customize the underlying SQL query. In order to do so, one can use the fact that `retrieve` methods typically return an actual result or value as well as the sqlalchemy selections that led to said result or value. We can use these selections and compile them to a standard, textual SQL query:

```
values, selections = req[0].retrieve(engine, constraint.ref)
print(str(selections[0].compile(engine, compile_kwargs={"literal_binds": True}))
```

In the case from above, this would return the following query:

```
SELECT
    anon_1.col_int,
    anon_1.col_varchar,
    count(*) AS count_1
FROM
    (SELECT
        tempdb.dbo.table.col_int AS col_int,
        tempdb.dbo.table.col_varchar AS col_varchar
    FROM
        tempdb.dbo.table WITH (NOLOCK)
    WHERE col_int >= 10) AS anon_1
GROUP BY anon_1.col_int, anon_1.col_varchar
```

1.8.3 Example 3: Comparing column structure

While we often care about value tuples of given columns, i.e. rows, it can also provide meaningful insights to compare the column structure of two tables. In particular, we might want to compare whether columns of one table are a subset or superset of another table. Moreover, for columns present in both tables, we'd like to learn about their respective types.

In order to illustrate such an example, we will again assume that there are two tables called `table1` and `table2`, irrespective of prior examples.

We can now create a `BetweenRequirement` for these two tables and use the `ColumnSubset Constraint`. As before, we will rely on the `get_factual_value` method to retrieve the values of interest for the first table passed to the `BetweenRequirement` and the `get_target_value` method for the second table passed to the `BetweenRequirement`.

```
import sqlalchemy as sa
from datajudge import BetweenRequirement

engine = sa.create_engine(your_connection_string)

req = BetweenRequirement.from_tables(
    db_name,
    schema_name,
    "table1",
    db_name,
    schema_name,
    "table2",
)

req.add_column_subset_constraint()

columns1 = req[0].get_factual_value(engine)
columns2 = req[0].get_target_value(engine)

print(f"Columns present in both: {set(columns1) & set(columns2)}")
```

(continues on next page)

(continued from previous page)

```
print(f"Columns present in only table1: {set(columns1) - set(columns2)}")
print(f"Columns present in only table2: {set(columns2) - set(columns1)}")
```

This could, for instance result in the following printout:

```
Columns present in both: {'col_varchar', 'col_int'}
Columns present in only table1: set()
Columns present in only table2: {'col_date'}
```

Now, we can investigate the types of the columns present in both tables:

```
for column in set(columns1) & set(columns2):
    req.add_column_type_constraint(column1=column, column2=column)
    type1 = req[0].get_factual_value(engine)
    type2 = req[0].get_target_value(engine)
    print(f"Column '{column}' has type '{type1}' in table1 and type '{type2}' in table2.
↪")
```

Depending on the underlying database management system and data, the output of this could for instance be:

```
Column 'col_varchar' has type 'varchar' in table1 and type 'varchar' in table2.
Column 'col_int' has type 'integer' in table1 and type 'integer' in table2.
```

1.9 Development

In order to work on `datajudge`, you can create a development conda environment as follows:

```
git clone https://github.com/Quantco/datajudge
cd datajudge
mamba env create
conda activate datajudge
pip install --no-build-isolation --disable-pip-version-check -e .
```

Unit tests can be run by executing

```
pytest tests/unit
```

Integration tests are run against a specific backend at a time. As of now, we provide helper scripts to spin up either a Postgres or MSSQL backend. Our `environment.yml` does not include the necessary dependencies for these backends, so you will need to install them manually.

To run integration tests against Postgres, first start a docker container with a Postgres database:

```
./start_postgres.sh
```

In your current environment, install the `psycopg2` package. After this, you may execute integration tests as follows:

```
pytest tests/integration --backend=postgres
```

Analogously, for MSSQL, run

```
./start_mssql.sh
```

and

```
pytest tests/integration --backend=mssql-freetds
```

or

```
pytest tests/integration --backend=mssql
```

depending on the driver you are using.

1.10 datajudge package

1.10.1 Submodules

datajudge.formatter module

class datajudge.formatter.**AnsiColorFormatter**

Bases: *Formatter*

Methods

apply_formatting
fmt_str

apply_formatting(*code*, *inner*)

Parameters

- **code** (str) –
- **inner** (str) –

Return type
str

class datajudge.formatter.**Formatter**

Bases: ABC

Methods

apply_formatting(_, inner)

fmt_str

apply_formatting(_, *inner*)

Parameters

- **_** (str) –
- **inner** (str) –

Return type
str

fmt_str(*string*)

Parameters
string (str) –

Return type
str

datajudge.pytest_integration module

datajudge.pytest_integration.**collect_data_tests**(*requirements*)

Make a Pytest test case that checks all *requirements*.

Returns a function named *test_constraint* that is parametrized over all constraints in *requirements*. The function requires a *datajudge_engine* fixture that is a SQLAlchemy engine to be available.

Parameters
requirements (Iterable[*Requirement*]) –

datajudge.pytest_integration.**get_formatter**(*pytestconfig*)

datajudge.utils module

datajudge.utils.**format_difference**(*n1*, *n2*, *decimal_separator=True*)

Given two numbers, n1 and n2, return a tuple of two strings, each representing one of the input numbers with the differing part highlighted. Highlighting is done using BBCode-like tags, which are replaced by the formatter.

Examples:
123, 123.0 -> 123, 123[numDiff].0[/numDiff] 122593859432, 122593859432347 -> 122593859432, 122593859432[numDiff]347[/numDiff]

Args: - n1: The first number to compare. - n2: The second number to compare. - decimal_separator: Whether to separate the decimal part of the numbers with commas.

Returns: - A tuple of two strings, each representing one of the input numbers with the differing part highlighted.

Parameters

- **n1** (Union[float, int]) –
- **n2** (Union[float, int]) –
- **decimal_separator** (bool) –

Return type
Tuple[str, str]

1.10.2 Module contents

datajudge allows to assess whether data from database complies with reference information.

class datajudge.**BetweenRequirement**(*data_source*, *data_source2*, *date_column=None*, *date_column2=None*)

Bases: *Requirement*

Methods

add_column_subset_constraint([name])	Columns of first table are subset of second table.
add_column_superset_constraint([name])	Columns of first table are superset of columns of second table.
add_column_type_constraint(column1, column2)	Check that the columns have the same type.
add_date_max_constraint(column1, column2[, ...])	Compare date max of first table to date max of second table.
add_date_min_constraint(column1, column2[, ...])	Ensure date min of first table is greater or equal date min of second table.
add_ks_2sample_constraint(column1, column2)	Apply the so-called two-sample Kolmogorov-Smirnov test to the distributions of the two given columns.
add_max_null_fraction_constraint(column1, ...)	Assert that the fraction of `NULL` values of one is at most that of the other.
add_n_rows_max_gain_constraint([...])	#rows from first table <= #rows from second table * (1 + max_growth).
add_n_rows_max_loss_constraint([...])	#rows from first table >= #rows from second table * (1 - max_loss).
add_n_rows_min_gain_constraint([...])	#rows from first table >= #rows from second table * (1 + min_growth).
add_n_uniques_max_gain_constraint(columns1 ...)	#uniques or first table <= #uniques of second table* (1 + max_growth).
add_n_uniques_max_loss_constraint(columns1 ...)	#uniques in first table <= #uniques in second table * (1 - max_loss).
add_numeric_percentile_constraint(column1, ...)	Assert that the `percentage`-th percentile is approximately equal.
add_row_equality_constraint(columns1, ...[, ...])	At most `max_missing_fraction` of rows in T1 and T2 are absent in either.
add_row_matching_equality_constraint(...[, ...])	Match tables in matching_columns, compare for equality in comparison_columns.
add_row_subset_constraint(columns1, ...[, ...])	At most `max_missing_fraction` of rows in T1 are not in T2.
add_row_superset_constraint(columns1, ...[, ...])	At most `max_missing_fraction` of rows in T2 are not in T1.
add_uniques_equality_constraint(columns1, ...)	Check if the data's unique values in given columns are equal.
add_uniques_subset_constraint(columns1, columns2)	Check if the given columns's unique values in are contained in reference data.
add_uniques_superset_constraint(columns1, ...)	Check if unique values of columns are contained in the reference data.
append(value)	S.append(value) -- append value to the end of the sequence

continues on next page

Table 7 – continued from previous page

clear()	
count(value)	
extend(values)	S.extend(iterable) -- extend sequence by appending elements from the iterable
from_expressions(expression1, expression2, ...)	Create a BetweenTableRequirement based on sqlalchemy expressions.
from_raw_queries(query1, query2, name1, name2)	Create a BetweenRequirement based on raw query strings.
index(value, [start, [stop]])	Raises ValueError if the value is not present.
insert(index, value)	S.insert(index, value) -- insert value before index
pop([index])	Raise IndexError if list is empty or index is out of range.
remove(value)	S.remove(value) -- remove first occurrence of value.
reverse()	S.reverse() -- reverse *IN PLACE*

add_n_rows_equality_constraint
add_n_uniques_equality_constraint
add_numeric_max_constraint
add_numeric_mean_constraint
add_numeric_min_constraint
add_varchar_max_length_constraint
add_varchar_min_length_constraint
from_tables
get_date_growth_rate
get_deviation_getter
test

Parameters

- **data_source** (DataSource) –
- **data_source2** (DataSource) –
- **date_column** (Optional[str]) –
- **date_column2** (Optional[str]) –

add_column_subset_constraint(*name=None*)

Columns of first table are subset of second table.

Parameters

name (str) –

add_column_superset_constraint(*name=None*)

Columns of first table are superset of columns of second table.

Parameters

name (str) –

add_column_type_constraint(*column1*, *column2*, *name=None*)

Check that the columns have the same type.

Parameters

- **column1** (str) –
- **column2** (str) –
- **name** (str) –

add_date_max_constraint(*column1*, *column2*, *use_upper_bound_reference=True*, *column_type='date'*, *condition1=None*, *condition2=None*, *name=None*)

Compare date max of first table to date max of second table.

The used columns of both tables need to be of the same type.

For more information on `column_type` values, see `add_column_type_constraint`.

If `use_upper_bound_reference`, the max of the first table has to be smaller or equal to the max of the second table. If not `use_upper_bound_reference`, the max of the first table has to be greater or equal to the max of the second table.

Parameters

- **column1** (str) –
- **column2** (str) –
- **use_upper_bound_reference** (bool) –
- **column_type** (Union[str, TypeEngine]) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_date_min_constraint(*column1*, *column2*, *use_lower_bound_reference=True*, *column_type='date'*, *condition1=None*, *condition2=None*, *name=None*)

Ensure date min of first table is greater or equal date min of second table.

The used columns of both tables need to be of the same type.

For more information on `column_type` values, see `add_column_type_constraint`.

If `use_lower_bound_reference`, the min of the first table has to be greater or equal to the min of the second table. If not `use_upper_bound_reference`, the min of the first table has to be smaller or equal to the min of the second table.

Parameters

- **column1** (str) –
- **column2** (str) –
- **use_lower_bound_reference** (bool) –
- **column_type** (Union[str, TypeEngine]) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_ks_2sample_constraint(*column1*, *column2*, *condition1=None*, *condition2=None*, *name=None*, *significance_level=0.05*)

Apply the so-called two-sample Kolmogorov-Smirnov test to the distributions of the two given columns. The constraint is fulfilled, when the resulting p-value of the test is higher than the significance level (default is 0.05, i.e., 5%). The signifance_level must be a value between 0.0 and 1.0.

Parameters

- **column1** (str) –
- **column2** (str) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –
- **significance_level** (float) –

add_max_null_fraction_constraint(*column1*, *column2*, *max_relative_deviation*, *condition1=None*, *condition2=None*, *name=None*)

Assert that the fraction of NULL values of one is at most that of the other.

Given that column2's underlying data has a fraction q of NULL values, the max_relative_deviation parameter allows column1's underlying data to have a fraction (1 + max_relative_deviation) * q of NULL values.

Parameters

- **column1** (str) –
- **column2** (str) –
- **max_relative_deviation** (float) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_n_rows_equality_constraint(*condition1=None*, *condition2=None*, *name=None*)

Parameters

- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_n_rows_max_gain_constraint(*constant_max_relative_gain=None*, *date_range_gain_deviation=None*, *condition1=None*, *condition2=None*, *name=None*)

#rows from first table <= #rows from second table * (1 + max_growth).

See readme for more information on max_growth.

Parameters

- **constant_max_relative_gain** (Optional[float]) –
- **date_range_gain_deviation** (Optional[float]) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_n_rows_max_loss_constraint(*constant_max_relative_loss=None*, *date_range_loss_deviation=None*, *condition1=None*, *condition2=None*, *name=None*)

#rows from first table >= #rows from second table * (1 - max_loss).

See readme for more information on max_loss.

Parameters

- **constant_max_relative_loss** (Optional[float]) –
- **date_range_loss_deviation** (Optional[float]) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_n_rows_min_gain_constraint(*constant_min_relative_gain=None*, *date_range_gain_deviation=None*, *condition1=None*, *condition2=None*, *name=None*)

#rows from first table >= #rows from second table * (1 + min_growth).

See readme for more information on min_growth.

Parameters

- **constant_min_relative_gain** (Optional[float]) –
- **date_range_gain_deviation** (Optional[float]) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_n_uniques_equality_constraint(*columns1*, *columns2*, *condition1=None*, *condition2=None*, *name=None*)

Parameters

- **columns1** (Optional[List[str]]) –
- **columns2** (Optional[List[str]]) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_n_uniques_max_gain_constraint(*columns1*, *columns2*, *constant_max_relative_gain=None*, *date_range_gain_deviation=None*, *condition1=None*, *condition2=None*, *name=None*)

#uniques or first table <= #uniques of second table* (1 + max_growth).

#uniques in first table are defined based on columns1, #uniques in second table are defined based on columns2.

See readme for more information on max_growth.

Parameters

- **columns1** (Optional[List[str]]) –
- **columns2** (Optional[List[str]]) –

- **constant_max_relative_gain** (Optional[float]) –
- **date_range_gain_deviation** (Optional[float]) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_n_uniques_max_loss_constraint(*columns1*, *columns2*, *constant_max_relative_loss=None*, *date_range_loss_deviation=None*, *condition1=None*, *condition2=None*, *name=None*)

#uniques in first table <= #uniques in second table * (1 - max_loss).

#uniques in first table are defined based on columns1, #uniques in second table are defined based on columns2.

See readme for more information on max_loss.

Parameters

- **columns1** (Optional[List[str]]) –
- **columns2** (Optional[List[str]]) –
- **constant_max_relative_loss** (Optional[float]) –
- **date_range_loss_deviation** (Optional[float]) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_numeric_max_constraint(*column1*, *column2*, *condition1=None*, *condition2=None*, *name=None*)

Parameters

- **column1** (str) –
- **column2** (str) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_numeric_mean_constraint(*column1*, *column2*, *max_absolute_deviation*, *condition1=None*, *condition2=None*, *name=None*)

Parameters

- **column1** (str) –
- **column2** (str) –
- **max_absolute_deviation** (float) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_numeric_min_constraint(*column1*, *column2*, *condition1=None*, *condition2=None*, *name=None*)

Parameters

- **column1** (str) –
- **column2** (str) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_numeric_percentile_constraint(*column1*, *column2*, *percentage*, *max_absolute_deviation=None*, *max_relative_deviation=None*, *condition1=None*, *condition2=None*, *name=None*)

Assert that the `percentage`-th percentile is approximately equal.

The percentile is defined as the value present in `column1` / `column2` for which `percentage` % of the values in `column1` / `column2` are less or equal. `NULL` values are ignored.

Hence, if `percentage` is less than the inverse of the number of non-`NULL` rows, `None` is received as the `percentage`-th percentile.

`percentage` is expected to be provided in percent. The median, for example, would correspond to `percentage=50`.

At least one of `max_absolute_deviation` and `max_relative_deviation` must be provided.

Parameters

- **column1** (str) –
- **column2** (str) –
- **percentage** (float) –
- **max_absolute_deviation** (Optional[float]) –
- **max_relative_deviation** (Optional[float]) –
- **condition1** (Optional[*Condition*]) –
- **condition2** (Optional[*Condition*]) –
- **name** (str) –

add_row_equality_constraint(*columns1*, *columns2*, *max_missing_fraction*, *condition1=None*, *condition2=None*, *name=None*)

At most `max_missing_fraction` of rows in T1 and T2 are absent in either.

In other words, $\frac{|T1-T2|+|T2-T1|}{|T1 \cup T2|} \leq$ `max_missing_fraction`. Rows from T1 are indexed in `columns1`, rows from T2 are indexed in `columns2`.

Parameters

- **columns1** (Optional[List[str]]) –
- **columns2** (Optional[List[str]]) –
- **max_missing_fraction** (float) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_row_matching_equality_constraint(*matching_columns1*, *matching_columns2*, *comparison_columns1*, *comparison_columns2*, *max_missing_fraction*, *condition1=None*, *condition2=None*, *name=None*)

Match tables in matching_columns, compare for equality in comparison_columns.

This constraint is similar to the nature of the `RowEquality` constraint. Just as the latter, this constraint divides the cardinality of an intersection by the cardinality of a union. The difference lies in how the set are created. While `RowEquality` considers all rows of both tables, indexed in columns, `RowMatchingEquality` considers only rows in both tables having values in `matching_columns` present in both tables. At most `max_missing_fraction` of such rows can be missing in the intersection.

Alternatively, this can be thought of as counting mismatches in `comparison_columns` after performing an inner join on `matching_columns`.

Parameters

- **matching_columns1** (`List[str]`) –
- **matching_columns2** (`List[str]`) –
- **comparison_columns1** (`List[str]`) –
- **comparison_columns2** (`List[str]`) –
- **max_missing_fraction** (`float`) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (`str`) –

add_row_subset_constraint(*columns1*, *columns2*, *constant_max_missing_fraction*, *date_range_loss_fraction=None*, *condition1=None*, *condition2=None*, *name=None*)

At most `max_missing_fraction` of rows in T1 are not in T2.

In other words, $\frac{|T1-T2|}{|T1|} \leq$ `max_missing_fraction`. Rows from T1 are indexed in columns1, rows from T2 are indexed in `columns2`.

In particular, the operation `|T1-T2|` relies on a sql `EXCEPT` statement. In contrast to `EXCEPT ALL`, this should lead to a set subtraction instead of a multiset subtraction. In other words, duplicates in T1 are treated as single occurrences.

Parameters

- **columns1** (`Optional[List[str]]`) –
- **columns2** (`Optional[List[str]]`) –
- **constant_max_missing_fraction** (`Optional[float]`) –
- **date_range_loss_fraction** (`Optional[float]`) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (`str`) –

add_row_superset_constraint(*columns1*, *columns2*, *constant_max_missing_fraction*, *date_range_loss_fraction=None*, *condition1=None*, *condition2=None*, *name=None*)

At most `max_missing_fraction` of rows in T2 are not in T1.

In other words, $\frac{|T2-T1|}{|T2|} \leq$ `max_missing_fraction`. Rows from T1 are indexed in `columns1`, rows from T2 are indexed in `columns2`.

Parameters

- **columns1** (`Optional[List[str]]`) –
- **columns2** (`Optional[List[str]]`) –
- **constant_max_missing_fraction** (`float`) –
- **date_range_loss_fraction** (`Optional[float]`) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (`str`) –

add_uniques_equality_constraint(*columns1*, *columns2*, *map_func=None*, *reduce_func=None*, *condition1=None*, *condition2=None*, *name=None*)

Check if the data's unique values in given columns are equal.

The `UniquesEquality` constraint asserts if the values contained in a column of a `DataSource`'s columns, are strictly the ones of another `DataSource`'s columns.

See the `Uniques` class for further parameter details on `map_func` and `reduce_func`.

Parameters

- **columns1** (`List[str]`) –
- **columns2** (`List[str]`) –
- **map_func** (`Callable[[TypeVar(T)], TypeVar(T)]`) –
- **reduce_func** (`Callable[[Collection], Collection]`) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (`str`) –

add_uniques_subset_constraint(*columns1*, *columns2*, *max_relative_violations=0*, *map_func=None*, *reduce_func=None*, *condition1=None*, *condition2=None*, *name=None*)

Check if the given columns's unique values in are contained in reference data.

The `UniquesSubset` constraint asserts if the values contained in given column of a `DataSource` are part of the unique values of given columns of another `DataSource`.

Null values in the column are ignored. To assert the non-existence of them use the `NullAbsence` constraint via the `add_null_absence_constraint` helper method for `WithinRequirement`.

`max_relative_violations` indicates what fraction of rows of the given table may have values not included in the reference set of unique values. Please note that `UniquesSubset` and `UniquesSuperset` are not symmetrical in this regard.

See `Uniques` for further details on `map_func` and `reduce_func`.

Parameters

- **columns1** (`List[str]`) –

- **columns2** (List[str]) –
- **max_relative_violations** (float) –
- **map_func** (Callable[[TypeVar(T)], TypeVar(T)]) –
- **reduce_func** (Callable[[Collection], Collection]) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_uniques_superset_constraint(*columns1*, *columns2*, *max_relative_violations=0*, *map_func=None*, *reduce_func=None*, *condition1=None*, *condition2=None*, *name=None*)

Check if unique values of columns are contained in the reference data.

The UniquesSuperset constraint asserts that reference set of expected values, derived from the unique values in given columns of the reference DataSource, is contained in given columns of a DataSource.

Null values in the column are ignored. To assert the non-existence of them use the NullAbsence constraint via the add_null_absence_constraint helper method for WithinRequirement.

max_relative_violations indicates what fraction of unique values of the given DataSource are not represented in the reference set of unique values. Please note that UniquesSubset and UniquesSuperset are not symmetrical in this regard.

One use of this constraint is to test for consistency in columns with expected categorical values.

See Uniques for further details on map_func and reduce_func.

Parameters

- **columns1** (List[str]) –
- **columns2** (List[str]) –
- **max_relative_violations** (float) –
- **map_func** (Callable[[TypeVar(T)], TypeVar(T)]) –
- **reduce_func** (Callable[[Collection], Collection]) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_varchar_max_length_constraint(*column1*, *column2*, *condition1=None*, *condition2=None*, *name=None*)

Parameters

- **column1** (str) –
- **column2** (str) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

add_varchar_min_length_constraint(*column1*, *column2*, *condition1=None*, *condition2=None*, *name=None*)

Parameters

- **column1** (str) –
- **column2** (str) –
- **condition1** (*Condition*) –
- **condition2** (*Condition*) –
- **name** (str) –

classmethod from_expressions(*expression1*, *expression2*, *name1*, *name2*, *date_column=None*, *date_column2=None*)

Create a BetweenTableRequirement based on sqlalchemy expressions.

Any sqlalchemy object implementing the alias method can be passed as an argument for the expression1 and expression2 parameters. This could, e.g. be a sqlalchemy.Table object or the result of a sqlalchemy.select invocation.

name1 and name2 will be used to represent the expressions in error messages, respectively.

Parameters

- **name1** (str) –
- **name2** (str) –
- **date_column** (Optional[str]) –
- **date_column2** (Optional[str]) –

classmethod from_raw_queries(*query1*, *query2*, *name1*, *name2*, *columns1=None*, *columns2=None*, *date_column=None*, *date_column2=None*)

Create a BetweenRequirement based on raw query strings.

The query1 and query2 parameters can be passed any query string returning rows, e.g. "SELECT * FROM myschema.mytable LIMIT 1337" or "SELECT id, name FROM table1 UNION SELECT id, name FROM table2".

name1 and name2 will be used to represent the queries in error messages, respectively.

If constraints rely on specific columns, these should be provided here via columns1 and columns2 respectively.

Parameters

- **query1** (str) –
- **query2** (str) –
- **name1** (str) –
- **name2** (str) –
- **columns1** (List[str]) –
- **columns2** (List[str]) –
- **date_column** (Optional[str]) –
- **date_column2** (Optional[str]) –

classmethod from_tables(*db_name1*, *schema_name1*, *table_name1*, *db_name2*, *schema_name2*, *table_name2*, *date_column=None*, *date_column2=None*)

Parameters

- **db_name1** (str) –
- **schema_name1** (str) –
- **table_name1** (str) –
- **db_name2** (str) –
- **schema_name2** (str) –
- **table_name2** (str) –
- **date_column** (Optional[str]) –
- **date_column2** (Optional[str]) –

get_date_growth_rate(*engine*)

Return type
float

get_deviation_getter(*fix_value*, *deviation*)

Parameters

- **fix_value** (Optional[float]) –
- **deviation** (Optional[float]) –

class datajudge.**Condition**(*raw_string=None*, *conditions=None*, *reduction_operator=None*)

Bases: object

Condition allows for further narrowing down of a DataSource in a Constraint.

A *Condition* can be thought of as a filter, the content of a sql 'where' clause or a condition as known from probability theory.

While a *DataSource* is expressed more generally, one might be interested in testing properties of a specific part of said *DataSource* in light of a particular constraint. Hence using *Condition`s allows for the reusage of a `DataSource, in lieu of creating a new custom `DataSource* with the *Condition* implicitly built in.

A *Condition* can either be 'atomic', i.e. not further reducible to sub-conditions or 'composite', i.e. combining multiple subconditions. In the former case, it can be instantiated with help of the *raw_string* parameter, e.g. *"col1 > 0"*. In the latter case, it can be instantiated with help of the *conditions* and *reduction_operator* parameters. *reduction_operator* allows for two values: *"and"* (logical conjunction) and *"or"* (logical disjunction). Note that composition of `Condition`s supports arbitrary degrees of nesting.

Attributes

conditions
raw_string
reduction_operator

Methods

snowflake_str

Parameters

- **raw_string** (str | None) –
- **conditions** (Optional[Sequence[*Condition*]]) –
- **reduction_operator** (str | None) –

conditions: Optional[Sequence[*Condition*]] = None

raw_string: str | None = None

reduction_operator: str | None = None

snowflake_str()

class datajudge.**Constraint**(*ref*, *, *ref2=None*, *ref_value=None*, *name=None*)

Bases: ABC

Express a DataReference constraint against either another DataReference or a reference value.

Constraints against other DataReferences are typically referred to as 'between' constraints. Please use the the *ref2* argument to instantiate such a constraint. Constraints against a fixed reference value are typically referred to as 'within' constraints. Please use the *ref_value* argument to instantiate such a constraint.

A constraint typically relies on the comparison of factual and target values. The former represent the key quantity of interest as seen in the database, the latter the key quantity of interest as expected a priori. Such a comparison is meant to be carried out in the *test* method.

In order to obtain such values, the *retrieve* method defines a mapping from DataReference, be it the DataReference of primary interest, *ref*, or a baseline DataReference, *ref2*, to value. If *ref_value* is already provided, usually no further mapping needs to be taken care of.

Attributes

condition_string
target_prefix

Methods

retrieve(engine, ref)	Retrieve the value of interest for a DataReference from database.

compare
get_description
get_factual_value
get_target_value
test

Parameters

- **ref** (DataReference) –
- **ref_value** (Any) –
- **name** (str) –

compare(*value_factual*, *value_target*)

Parameters

- **value_factual** (Any) –
- **value_target** (Any) –

Return type
Tuple[bool, Optional[str]]

property condition_string: str

get_description()

Return type
str

get_factual_value(*engine*)

Parameters
engine (Engine) –

Return type
Any

get_target_value(*engine*)

Parameters
engine (Engine) –

Return type
Any

retrieve(*engine*, *ref*)

Retrieve the value of interest for a DataReference from database.

Parameters

- **engine** (Engine) –
- **ref** (DataReference) –

Return type
Tuple[Any, Optional[List[Select]]]

property target_prefix: str

test(*engine*)

Parameters
engine (Engine) –

Return type
TestResult

class datajudge.**Requirement**

Bases: ABC, MutableSequence

Methods

append(value)	S.append(value) -- append value to the end of the sequence
clear()	
count(value)	
extend(values)	S.extend(iterable) -- extend sequence by appending elements from the iterable
index(value, [start, [stop]])	Raises ValueError if the value is not present.
insert(index, value)	S.insert(index, value) -- insert value before index
pop([index])	Raise IndexError if list is empty or index is out of range.
remove(value)	S.remove(value) -- remove first occurrence of value.
reverse()	S.reverse() -- reverse *IN PLACE*

test

insert(*index*, *value*)

S.insert(index, value) – insert value before index

Parameters

- **index** (int) –
- **value** (*Constraint*) –

Return type
None

test(*engine*)

Return type
List[TestResult]

class datajudge.**WithinRequirement**(*data_source*)

Bases: *Requirement*

Methods

add_categorical_bound_constraint(columns, ...)	Check if the distribution of unique values in columns falls within the specified minimum and maximum bounds.
add_column_type_constraint(column, column_type)	Check if a column type matches the expected column_type.
add_date_between_constraint(column, ...[, ...])	Use string format: lower_bound="'20121230'".
add_date_max_constraint(column, max_value[, ...])	Ensure all dates to be superior than max_value.
add_date_min_constraint(column, min_value[, ...])	Ensure all dates to be superior than min_value.

continues on next page

Table 8 – continued from previous page

add_date_no_gap_constraint(start_column, ...)	Express that date range rows have no gap in-between them.
add_date_no_overlap_2d_constraint(...[, ...])	Express that several date range rows do not overlap in two date dimensions.
add_date_no_overlap_constraint(start_column, ...)	Constraint expressing that several date range rows may not overlap.
add_functional_dependency_constraint(...[, ...])	Expresses a functional dependency, a constraint where the *value_columns* are uniquely determined by the *key_columns*.
add_groupby_aggregation_constraint(columns ...)	Check whether array aggregate corresponds to an integer range.
add_max_null_fraction_constraint(column, ...)	Assert that `column` has less than a certain fraction of `NULL` values.
add_numeric_between_constraint(column, ...)	Assert that the column's values lie between `lower_bound` and `upper_bound`.
add_numeric_max_constraint(column, max_value)	All values in column are less or equal max_value.
add_numeric_mean_constraint(column, ...[, ...])	Assert the mean of the column deviates at most max_deviation from mean_value.
add_numeric_min_constraint(column, min_value)	All values in column are greater or equal min_value.
add_numeric_no_gap_constraint(start_column, ...)	Express that numeric interval rows have no gaps larger than some max value in-between them.
add_numeric_no_overlap_constraint(...[, ...])	Constraint expressing that several numeric interval rows may not overlap.
add_numeric_percentile_constraint(column, ...)	Assert that the `percentage`-th percentile is approximately `expected_percentile`.
add_primary_key_definition_constraint(...[, ...])	Check that the primary key constraints in the database are exactly equal to the given column names.
add_uniqueness_constraint([columns, ...])	Columns should uniquely identify row.
add_uniques_equality_constraint(columns, uniques)	Check if the data's unique values are equal to a given set of values.
add_uniques_subset_constraint(columns, uniques)	Check if the data's unique values are contained in a given set of values.
add_uniques_superset_constraint(columns, uniques)	Check if unique values of columns are contained in the reference data.
add_varchar_regex_constraint(column, regex)	Assesses whether the values in a column match a given regular expression pattern.
add_varchar_regex_constraint_db(column, regex)	Assesses whether the values in a column match a given regular expression pattern.
append(value)	S.append(value) -- append value to the end of the sequence
clear()	
count(value)	
extend(values)	S.extend(iterable) -- extend sequence by appending elements from the iterable
from_expression(expression, name)	Create a `WithinRequirement` based on a sqlalchemy expression.
from_raw_query(query, name[, columns])	Create a `WithinRequirement` based on a raw query string.

continues on next page

Table 8 – continued from previous page

index(value, [start, [stop]])	Raises ValueError if the value is not present.
insert(index, value)	S.insert(index, value) -- insert value before index
pop([index])	Raise IndexError if list is empty or index is out of range.
remove(value)	S.remove(value) -- remove first occurrence of value.
reverse()	S.reverse() -- reverse *IN PLACE*

add_column_existence_constraint
add_n_rows_equality_constraint
add_n_rows_max_constraint
add_n_rows_min_constraint
add_n_uniques_equality_constraint
add_null_absence_constraint
add_varchar_max_length_constraint
add_varchar_min_length_constraint
from_table
test

Parameters

data_source (DataSource) –

add_categorical_bound_constraint(*columns*, *distribution*, *default_bounds=(0, 0)*, *max_relative_violations=0*, *condition=None*, *name=None*)

Check if the distribution of unique values in columns falls within the specified minimum and maximum bounds.

The *CategoricalBoundConstraint* is added to ensure the distribution of unique values in the specified columns of a *DataSource* falls within the given minimum and maximum bounds defined in the *distribution* parameter.

Parameters

columns
[List[str]] A list of column names from the *DataSource* to apply the constraint on.

distribution
[Dict[T, Tuple[float, float]]] A dictionary where keys represent unique values and the corresponding tuple values represent the minimum and maximum allowed proportions of the respective unique value in the columns.

default_bounds
[Tuple[float, float], optional, default=(0, 0)] A tuple specifying the minimum and maximum allowed proportions for all elements not mentioned in the distribution. By default, it's set to (0, 0), which means all elements not present in *distribution* will cause a constraint failure.

max_relative_violations
[float, optional, default=0] A tolerance threshold (0 to 1) for the proportion of elements in the data that can violate the bound constraints without triggering the constraint violation.

condition
[Condition, optional] An optional parameter to specify a *Condition* object to filter the data before applying the constraint.

name
[str, optional] An optional parameter to provide a custom name for the constraint.

Parameters

- **columns** (List[str]) –
- **distribution** (Dict[TypeVar(T), Tuple[float, float]]) –
- **default_bounds** (Tuple[float, float]) –
- **max_relative_violations** (float) –
- **condition** (*Condition*) –
- **name** (str) –

add_column_existence_constraint(*columns*, *name=None*)

Parameters

- **columns** (List[str]) –
- **name** (str) –

add_column_type_constraint(*column*, *column_type*, *name=None*)

Check if a column type matches the expected column_type.

The column_type can be provided as a string (backend-specific type name), a backend-specific SQLAlchemy type, or a SQLAlchemy's generic type.

If SQLAlchemy's generic types are used, the check is performed using *isinstance*, which means that the actual type can also be a subclass of the target type. For more information on SQLAlchemy's generic types, see https://docs.sqlalchemy.org/en/20/core/type_basics.html

Parameters

column
[str] The name of the column to which the constraint will be applied.

column_type
[Union[str, sa.types.TypeEngine]] The expected type of the column. This can be a string, a backend-specific SQLAlchemy type, or a generic SQLAlchemy type.

name
[Optional[str]] An optional name for the constraint. If not provided, a name will be generated automatically.

Parameters

- **column** (str) –
- **column_type** (Union[str, TypeEngine]) –
- **name** (str) –

add_date_between_constraint(*column*, *lower_bound*, *upper_bound*, *min_fraction*, *condition=None*, *name=None*)

Use string format: lower_bound="'20121230'".

Parameters

- **column** (str) –
- **lower_bound** (str) –
- **upper_bound** (str) –

- **min_fraction** (float) –
- **condition** (*Condition*) –
- **name** (str) –

add_date_max_constraint(*column*, *max_value*, *use_upper_bound_reference=True*, *column_type='date'*, *condition=None*, *name=None*)

Ensure all dates to be superior than max_value.

Use string format: max_value="'20121230'".

For more information on `column_type` values, see `add_column_type_constraint`.

If `use_upper_bound_reference`, the max of the first table has to be smaller or equal to `max_value`. If not `use_upper_bound_reference`, the max of the first table has to be greater or equal to `max_value`.

Parameters

- **column** (str) –
- **max_value** (str) –
- **use_upper_bound_reference** (bool) –
- **column_type** (Union[str, TypeEngine]) –
- **condition** (*Condition*) –
- **name** (str) –

add_date_min_constraint(*column*, *min_value*, *use_lower_bound_reference=True*, *column_type='date'*, *condition=None*, *name=None*)

Ensure all dates to be superior than min_value.

Use string format: min_value="'20121230'".

For more information on `column_type` values, see `add_column_type_constraint`.

If `use_lower_bound_reference`, the min of the first table has to be greater or equal to `min_value`. If not `use_upper_bound_reference`, the min of the first table has to be smaller or equal to `min_value`.

Parameters

- **column** (str) –
- **min_value** (str) –
- **use_lower_bound_reference** (bool) –
- **column_type** (Union[str, TypeEngine]) –
- **condition** (*Condition*) –
- **name** (str) –

add_date_no_gap_constraint(*start_column*, *end_column*, *key_columns=None*, *end_included=True*, *max_relative_n_violations=0*, *condition=None*, *name=None*)

Express that date range rows have no gap in-between them.

The table under inspection must consist of at least one but up to many key columns, identifying an entity. Additionally, a `start_column` and an `end_column`, indicating start and end dates, should be provided.

Neither of those columns should contain `NULL` values. Also, it should hold that for a given row, the value of `end_column` is strictly greater than the value of `start_column`.

Note that the value of start_column is expected to be included in each date range. By default, the value of end_column is expected to be included as well - this can however be changed by setting end_included to False.

A 'key' is a fixed set of values in key_columns and represents an entity of interest. A priori, a key is not a primary key, i.e., a key can have and often has several rows. Thereby, a key will often come with several date ranges.

If`` key_columns`` is None or [], all columns of the table will be considered as composing the key.

In order to express a tolerance for some violations of this gap property, use the max_relative_n_violations parameter. The latter expresses for what fraction of all key_values, at least one gap may exist.

For illustrative examples of this constraint, please refer to its test cases.

Parameters

- **start_column** (str) –
- **end_column** (str) –
- **key_columns** (Optional[List[str]]) –
- **end_included** (bool) –
- **max_relative_n_violations** (float) –
- **condition** (*Condition*) –
- **name** (str) –

add_date_no_overlap_2d_constraint(*start_column1*, *end_column1*, *start_column2*, *end_column2*, *key_columns=None*, *end_included=True*, *max_relative_n_violations=0*, *condition=None*, *name=None*)

Express that several date range rows do not overlap in two date dimensions.

The table under inspection must consist of at least one but up to many key columns, identifying an entity. Per date dimension, a start_column and an end_column should be provided.

For a given row in this table, start_column1 and end_column1 indicate a date range. Moreoever, for that same row, start_column2 and end_column2 indicate a date range. These date ranges are expected to represent different date 'dimensions'. Example: A row indicates a forecasted value used in production. start_column1 and end_column1 represent the timespan that was forecasted, e.g. the weather from next Saturday to next Sunday. end_column1 and end_column2 might indicate the timespan when this forceast was used, e.g. from the previous Monday to Wednesday.

Neither of those columns should contain NULL values. Also it should hold that for a given row, the value of end_column is strictly greater than the value of start_column.

Note that the values of start_column1 and start_column2 are expected to be included in each date range. By default, the values of end_column1 and end_column2 are expected to be included as well - this can however be changed by setting end_included to False.

A 'key' is a fixed set of values in key_columns and represents an entity of interest. A priori, a key is not a primary key, i.e., a key can have and often has several rows. Thereby, a key will often come with several date ranges.

Often, you might want the date ranges for a given key not to overlap.

If key_columns is None or [], all columns of the table will be considered as composing the key.

In order to express a tolerance for some violations of this non-overlapping property, use the `max_relative_n_violations` parameter. The latter expresses for what fraction of all key_values, at least one overlap may exist.

For illustrative examples of this constraint, please refer to its test cases.

Parameters

- **start_column1** (str) –
- **end_column1** (str) –
- **start_column2** (str) –
- **end_column2** (str) –
- **key_columns** (Optional[List[str]]) –
- **end_included** (bool) –
- **max_relative_n_violations** (float) –
- **condition** (*Condition*) –
- **name** (str) –

add_date_no_overlap_constraint(*start_column*, *end_column*, *key_columns=None*, *end_included=True*, *max_relative_n_violations=0*, *condition=None*, *name=None*)

Constraint expressing that several date range rows may not overlap.

The `DataSource` under inspection must consist of at least one but up to many `key_columns`, identifying an entity, a `start_column` and an `end_column`.

For a given row in this `DataSource`, `start_column` and `end_column` indicate a date range. Neither of those columns should contain NULL values. Also, it should hold that for a given row, the value of `end_column` is strictly greater than the value of `start_column`.

Note that the value of `start_column` is expected to be included in each date range. By default, the value of `end_column` is expected to be included as well - this can however be changed by setting `end_included` to `False`.

A 'key' is a fixed set of values in `key_columns` and represents an entity of interest. A priori, a key is not a primary key, i.e., a key can have and often has several rows. Thereby, a key will often come with several date ranges.

Often, you might want the date ranges for a given key not to overlap.

If `key_columns` is `None` or `[]`, all columns of the table will be considered as composing the key.

In order to express a tolerance for some violations of this non-overlapping property, use the `max_relative_n_violations` parameter. The latter expresses for what fraction of all key values, at least one overlap may exist.

For illustrative examples of this constraint, please refer to its test cases.

Parameters

- **start_column** (str) –
- **end_column** (str) –
- **key_columns** (Optional[List[str]]) –
- **end_included** (bool) –
- **max_relative_n_violations** (float) –

- **condition** (*Condition*) –
- **name** (str) –

add_functional_dependency_constraint(*key_columns*, *value_columns*, *condition=None*, *name=None*)

Expresses a functional dependency, a constraint where the *value_columns* are uniquely determined by the *key_columns*. This means that for each unique combination of values in the *key_columns*, there is exactly one corresponding combination of values in the *value_columns*.

The `add_unique_constraint` constraint is a special case of this constraint, where the *key_columns* are a primary key, and all other columns are included *value_columns*. This constraint allows for a more general definition of functional dependencies, where the *key_columns* are not necessarily a primary key.

For more information on functional dependencies, see https://en.wikipedia.org/wiki/Functional_dependency.

Parameters

- **key_columns** (List[str]) –
- **value_columns** (List[str]) –
- **condition** (*Condition*) –
- **name** (str) –

add_groupby_aggregation_constraint(*columns*, *aggregation_column*, *start_value*, *tolerance=0*, *condition=None*, *name=None*)

Check whether array aggregate corresponds to an integer range.

The `DataSource` is grouped by `columns`. Sql's `array_agg` function is then applied to the `aggregate_column`.

Since we expect `aggregate_column` to be a numeric column, this leads to a multiset of aggregated values. These values should correspond to the integers ranging from `start_value` to the cardinality of the multiset.

In order to allow for slight deviations from this pattern, `tolerance` expresses the fraction of all grouped-by rows, which may be incomplete ranges.

Parameters

- **columns** (Sequence[str]) –
- **aggregation_column** (str) –
- **start_value** (int) –
- **tolerance** (float) –
- **condition** (*Condition*) –
- **name** (str) –

add_max_null_fraction_constraint(*column*, *max_null_fraction*, *condition=None*, *name=None*)

Assert that `column` has less than a certain fraction of `NULL` values.

`max_null_fraction` is expected to lie within [0, 1].

Parameters

- **column** (str) –
- **max_null_fraction** (float) –
- **condition** (*Condition*) –

- **name** (str) –

add_n_rows_equality_constraint(*n_rows*, *condition=None*, *name=None*)

Parameters

- **n_rows** (int) –
- **condition** (*Condition*) –
- **name** (str) –

add_n_rows_max_constraint(*n_rows_max*, *condition=None*, *name=None*)

Parameters

- **n_rows_max** (int) –
- **condition** (*Condition*) –
- **name** (str) –

add_n_rows_min_constraint(*n_rows_min*, *condition=None*, *name=None*)

Parameters

- **n_rows_min** (int) –
- **condition** (*Condition*) –
- **name** (str) –

add_n_uniques_equality_constraint(*columns*, *n_uniques*, *condition=None*, *name=None*)

Parameters

- **columns** (Optional[List[str]]) –
- **n_uniques** (int) –
- **condition** (*Condition*) –
- **name** (str) –

add_null_absence_constraint(*column*, *condition=None*, *name=None*)

Parameters

- **column** (str) –
- **condition** (*Condition*) –
- **name** (str) –

add_numeric_between_constraint(*column*, *lower_bound*, *upper_bound*, *min_fraction*, *condition=None*, *name=None*)

Assert that the column's values lie between `lower_bound` and `upper_bound`.

Note that both bounds are inclusive.

Unless specified otherwise via the usage of a `condition`, `NULL` values will be considered in the denominator of `min_fraction`. `NULL` values will never be considered to lie in the interval [`lower_bound`, `upper_bound`].

Parameters

- **column** (str) –

- **lower_bound** (float) –
- **upper_bound** (float) –
- **min_fraction** (float) –
- **condition** (*Condition*) –
- **name** (str) –

add_numeric_max_constraint(*column*, *max_value*, *condition=None*, *name=None*)

All values in column are less or equal max_value.

Parameters

- **column** (str) –
- **max_value** (float) –
- **condition** (*Condition*) –
- **name** (str) –

add_numeric_mean_constraint(*column*, *mean_value*, *max_absolute_deviation*, *condition=None*, *name=None*)

Assert the mean of the column deviates at most max_deviation from mean_value.

Parameters

- **column** (str) –
- **mean_value** (float) –
- **max_absolute_deviation** (float) –
- **condition** (*Condition*) –
- **name** (str) –

add_numeric_min_constraint(*column*, *min_value*, *condition=None*)

All values in column are greater or equal min_value.

Parameters

- **column** (str) –
- **min_value** (float) –
- **condition** (*Condition*) –

add_numeric_no_gap_constraint(*start_column*, *end_column*, *key_columns=None*, *legitimate_gap_size=0*, *max_relative_n_violations=0*, *condition=None*, *name=None*)

Express that numeric interval rows have no gaps larger than some max value in-between them. The table under inspection must consist of at least one but up to many key columns, identifying an entity. Additionally, a `start_column` and an `end_column`, indicating interval start and end values, should be provided.

Neither of those columns should contain `NULL` values. Also, it should hold that for a given row, the value of `end_column` is strictly greater than the value of `start_column`.

`legitimate_gap_size` is the maximum tollerated gap size between two intervals.

A 'key' is a fixed set of values in `key_columns` and represents an entity of interest. A priori, a key is not a primary key, i.e., a key can have and often has several rows. Thereby, a key will often come with several intervals.

If`` key_columns`` is `None` or `[]`, all columns of the table will be considered as composing the key.

In order to express a tolerance for some violations of this gap property, use the `max_relative_n_violations` parameter. The latter expresses for what fraction of all key_values, at least one gap may exist.

For illustrative examples of this constraint, please refer to its test cases.

Parameters

- **start_column** (`str`) –
- **end_column** (`str`) –
- **key_columns** (`Optional[List[str]]`) –
- **legitimate_gap_size** (`float`) –
- **max_relative_n_violations** (`float`) –
- **condition** (*Condition*) –
- **name** (`str`) –

add_numeric_no_overlap_constraint(*start_column*, *end_column*, *key_columns=None*, *end_included=True*, *max_relative_n_violations=0*, *condition=None*, *name=None*)

Constraint expressing that several numeric interval rows may not overlap.

The `DataSource` under inspection must consist of at least one but up to many `key_columns`, identifying an entity, a `start_column` and an `end_column`.

For a given row in this `DataSource`, `start_column` and `end_column` indicate a numeric interval. Neither of those columns should contain NULL values. Also, it should hold that for a given row, the value of `end_column` is strictly greater than the value of `start_column`.

Note that the value of `start_column` is expected to be included in each interval. By default, the value of `end_column` is expected to be included as well - this can however be changed by setting `end_included` to `False`.

A 'key' is a fixed set of values in `key_columns` and represents an entity of interest. A priori, a key is not a primary key, i.e., a key can have and often has several rows. Thereby, a key will often come with several intervals.

Often, you might want the intervals for a given key not to overlap.

If `key_columns` is `None` or `[]`, all columns of the table will be considered as composing the key.

In order to express a tolerance for some violations of this non-overlapping property, use the `max_relative_n_violations` parameter. The latter expresses for what fraction of all key values, at least one overlap may exist.

For illustrative examples of this constraint, please refer to its test cases.

Parameters

- **start_column** (`str`) –
- **end_column** (`str`) –
- **key_columns** (`Optional[List[str]]`) –
- **end_included** (`bool`) –
- **max_relative_n_violations** (`float`) –
- **condition** (*Condition*) –

- **name** (str) –

add_numeric_percentile_constraint(*column*, *percentage*, *expected_percentile*, *max_absolute_deviation=None*, *max_relative_deviation=None*, *condition=None*, *name=None*)

Assert that the percentage-th percentile is approximately expected_percentile.

The percentile is defined as the value present in column for which percentage % of the values in column are less or equal. NULL values are ignored.

Hence, if percentage is less than the inverse of the number of non-NULL rows, None is received as the percentage -th percentile.

percentage is expected to be provided in percent. The median, for example, would correspond to percentage=50.

At least one of max_absolute_deviation and max_relative_deviation must be provided.

Parameters

- **column** (str) –
- **percentage** (float) –
- **expected_percentile** (float) –
- **max_absolute_deviation** (Optional[float]) –
- **max_relative_deviation** (Optional[float]) –
- **condition** (*Condition*) –
- **name** (str) –

add_primary_key_definition_constraint(*primary_keys*, *name=None*)

Check that the primary key constraints in the database are exactly equal to the given column names.

Note that this doesn't actually check that the primary key values are unique across the table.

Parameters

- **primary_keys** (List[str]) –
- **name** (str) –

add_uniqueness_constraint(*columns=None*, *max_duplicate_fraction=0*, *condition=None*, *max_absolute_n_duplicates=0*, *infer_pk_columns=False*, *name=None*)

Columns should uniquely identify row.

Given a set of columns, satisfy conditions of a primary key, i.e. uniqueness of tuples from said columns. This constraint has a tolerance for inconsistencies, expressed via max_duplicate_fraction. The latter suggests that the number of uniques from said columns is larger or equal to (1 - max_duplicate_fraction) the number of rows.

If infer_pk_columns is True, columns will be retrieved from the primary keys. When columns=None and infer_pk_columns=False, the fallback is validating that all rows in a table are unique.

Parameters

- **columns** (List[str]) –
- **max_duplicate_fraction** (float) –
- **condition** (*Condition*) –
- **max_absolute_n_duplicates** (int) –

- **infer_pk_columns** (bool) –
- **name** (str) –

add_uniques_equality_constraint(*columns*, *uniques*, *map_func=None*, *reduce_func=None*, *condition=None*, *name=None*)

Check if the data's unique values are equal to a given set of values.

The UniquesEquality constraint asserts if the values contained in a column of a DataSource are strictly the ones of a reference set of expected values, specified via the uniques parameter.

See the Uniques class for further parameter details on map_func and reduce_func.

Parameters

- **columns** (List[str]) –
- **uniques** (Collection[TypeVar(T)]) –
- **map_func** (Callable[[TypeVar(T)], TypeVar(T)]) –
- **reduce_func** (Callable[[Collection], Collection]) –
- **condition** (*Condition*) –
- **name** (str) –

add_uniques_subset_constraint(*columns*, *uniques*, *max_relative_violations=0*, *map_func=None*, *reduce_func=None*, *condition=None*, *name=None*)

Check if the data's unique values are contained in a given set of values.

The UniquesSubset constraint asserts if the values contained in a column of a DataSource are part of a reference set of expected values, specified via uniques.

Null values in the column are ignored. To assert the non-existence of them use the NullAbsence constraint via the add_null_absence_constraint helper method for WithinRequirement.

max_relative_violations indicates what fraction of rows of the given table may have values not included in the reference set of unique values. Please note that UniquesSubset and UniquesSuperset are not symmetrical in this regard.

See Uniques for further details on map_func and reduce_func.

Parameters

- **columns** (List[str]) –
- **uniques** (Collection[TypeVar(T)]) –
- **max_relative_violations** (float) –
- **map_func** (Callable[[TypeVar(T)], TypeVar(T)]) –
- **reduce_func** (Callable[[Collection], Collection]) –
- **condition** (*Condition*) –
- **name** (str) –

add_uniques_superset_constraint(*columns*, *uniques*, *max_relative_violations=0*, *map_func=None*, *reduce_func=None*, *condition=None*, *name=None*)

Check if unique values of columns are contained in the reference data.

The UniquesSuperset constraint asserts that reference set of expected values, specified via uniques, is contained in given columns of a DataSource.

Null values in the column are ignored. To assert the non-existence of them use the `NullAbsence` constraint via the `add_null_absence_constraint` helper method for `WithinRequirement`.

`max_relative_violations` indicates what fraction of unique values of the given `DataSource` are not represented in the reference set of unique values. Please note that `UniquesSubset` and `UniquesSuperset` are not symmetrical in this regard.

One use of this constraint is to test for consistency in columns with expected categorical values.

See `Uniques` for further details on `map_func` and `reduce_func`.

Parameters

- **columns** (`List[str]`) –
- **uniques** (`Collection[TypeVar(T)]`) –
- **max_relative_violations** (`float`) –
- **map_func** (`Callable[[TypeVar(T)], TypeVar(T)]`) –
- **reduce_func** (`Callable[[Collection], Collection]`) –
- **condition** (*Condition*) –
- **name** (`str`) –

add_varchar_max_length_constraint(*column*, *max_length*, *condition=None*, *name=None*)

Parameters

- **column** (`str`) –
- **max_length** (`int`) –
- **condition** (*Condition*) –
- **name** (`str`) –

add_varchar_min_length_constraint(*column*, *min_length*, *condition=None*, *name=None*)

Parameters

- **column** (`str`) –
- **min_length** (`int`) –
- **condition** (*Condition*) –
- **name** (`str`) –

add_varchar_regex_constraint(*column*, *regex*, *condition=None*, *name=None*, *allow_none=False*, *relative_tolerance=0.0*, *aggregated=True*, *n_counterexamples=5*)

Assesses whether the values in a column match a given regular expression pattern.

The option `allow_none` can be used in cases where the column is defined as nullable and contains null values.

How the tolerance factor is calculated can be controlled with the `aggregated` flag. When `True`, the tolerance is calculated using unique values. If not, the tolerance is calculated using all the instances of the data.

`n_counterexamples` defines how many counterexamples are displayed in an assertion text. If all counterexamples are meant to be shown, provide `-1` as an argument.

When using this method, the regex matching will take place in memory. If instead, you would like the matching to take place in database which is typically faster and substantially more memory-saving, please consider using `add_varchar_regex_constraint_db`.

Parameters

- **column** (`str`) –
- **regex** (`str`) –
- **condition** (*Condition*) –
- **name** (`str`) –
- **allow_none** (`bool`) –
- **relative_tolerance** (`float`) –
- **aggregated** (`bool`) –
- **n_counterexamples** (`int`) –

add_varchar_regex_constraint_db(*column*, *regex*, *condition=None*, *name=None*, *relative_tolerance=0.0*, *aggregated=True*, *n_counterexamples=5*)

Assesses whether the values in a column match a given regular expression pattern.

How the tolerance factor is calculated can be controlled with the `aggregated` flag. When `True`, the tolerance is calculated using unique values. If not, the tolerance is calculated using all the instances of the data.

`n_counterexamples` defines how many counterexamples are displayed in an assertion text. If all counterexamples are meant to be shown, provide `-1` as an argument.

When using this method, the regex matching will take place in database, which is only supported for Postgres, Sqllite and Snowflake. Note that for this feature is only for Snowflake when using sqlalchemy-snowflake >= 1.4.0. As an alternative, `add_varchar_regex_constraint` performs the regex matching in memory. This is typically slower and more expensive in terms of memory but available on all supported database mamangement systems.

Parameters

- **column** (`str`) –
- **regex** (`str`) –
- **condition** (*Condition*) –
- **name** (`str`) –
- **relative_tolerance** (`float`) –
- **aggregated** (`bool`) –
- **n_counterexamples** (`int`) –

classmethod from_expression(*expression*, *name*)

Create a `WithinRequirement` based on a sqlalchemy expression.

Any sqlalchemy object implementing the `alias` method can be passed as an argument for the `expression` parameter. This could, e.g. be an `sqlalchemy.Table` object or the result of a `sqlalchemy.select` call.

The `name` will be used to represent this expression in error messages.

Parameters

- **expression** (`FromClause`) –

- **name** (str) –

classmethod from_raw_query(*query*, *name*, *columns=None*)

Create a WithinRequirement based on a raw query string.

The query parameter can be passed any query string returning rows, e.g. "SELECT * FROM myschema.mytable LIMIT 1337" or "SELECT id, name FROM table1 UNION SELECT id, name FROM table2".

The name will be used to represent this query in error messages.

If constraints rely on specific columns, these should be provided here via columns, e.g. ["id", "name"].

Parameters

- **query** (str) –
- **name** (str) –
- **columns** (List[str]) –

classmethod from_table(*db_name*, *schema_name*, *table_name*)

Parameters

- **db_name** (str) –
- **schema_name** (str) –
- **table_name** (str) –

PYTHON MODULE INDEX

d

INDEX

A

B

C

D

F

G

I

M

R

S

T

W