great_expectations Documentation

The Great Expectations Team

Aug 11, 2020

CONTENTS

Great Expectations is a leading tool for *validating*, *documenting*, and *profiling*, your data to maintain quality and improve communication between teams. Head over to the *Introduction* to learn more, or jump straight to our *Getting started* guide.

CHAPTER

ONE

INTRODUCTION

Always know what to expect from your data.

1.1 What is Great Expectations?

Great Expectations helps teams save time and promote analytic integrity by offering a unique approach to automated testing: pipeline tests. Pipeline tests are applied to data (instead of code) and at batch time (instead of compile or deploy time). Pipeline tests are like unit tests for datasets: they help you guard against upstream data changes and monitor data quality.

Software developers have long known that automated testing is essential for managing complex codebases. Great Expectations brings the same discipline, confidence, and acceleration to data science and engineering teams.

1.2 Why would I use Great Expectations?

To get more done with data, faster. Teams use Great Expectations to

- Save time during data cleaning and munging.
- Accelerate ETL and data normalization.
- Streamline analyst-to-engineer handoffs.
- Streamline knowledge capture and requirements gathering from subject-matter experts.
- Monitor data quality in production data pipelines and data products.
- Automate verification of new data deliveries from vendors and other teams.
- Simplify debugging data pipelines if (when) they break.
- Codify assumptions used to build models when sharing with other teams or analysts.
- Develop rich, shared data documention in the course of normal work.
- Make implicit knowledge explicit.
- . . . and much more

1.3 Key features

Expectations

Expectations are the workhorse abstraction in Great Expectations. Like assertions in traditional python unit tests, Expectations provide a flexible, declarative language for describing expected behavior. Unlike traditional unit tests, Great Expectations applies Expectations to data instead of code.

Great Expectations currently supports native execution of Expectations in three environments: pandas, SQL (through the SQLAlchemy core), and Spark. This approach follows the philosophy of "take the compute to the data." Future releases of Great Expectations will extend this functionality to other frameworks, such as dask and BigQuery.

Automated data profiling

Writing pipeline tests from scratch can be tedious and counterintuitive. Great Expectations jump starts the process by providing powerful tools for automated data profiling. This provides the double benefit of helping you explore data faster, and capturing knowledge for future documentation and testing.

DataContexts and DataSources

...allow you to configure connections your data stores, using names you're already familiar with: "the ml_training_results bucket in S3," "the Users table in Redshift." Great Expectations provides convenience libraries to introspect most common data stores (Ex: SQL databases, data directories and S3 buckets.) We are also working to integrate with pipeline execution frameworks (Ex: Airflow, dbt, Dagster, Prefect). The Great Expectations framework lets you fetch, validate, profile, and document your data in a way that's meaningful within your existing infrastructure and work environment.

Tooling for validation

Evaluating Expectations against data is just one step in a typical validation workflow. Great Expectations makes the followup steps simple, too: storing validation results to a shared bucket, summarizing results and posting notifications to slack, handling differences between warnings and errors, etc.

Great Expectations also provides robust concepts of Batches and Runs. Although we sometimes talk informally about validating "dataframes" or "tables," it's much more common to validate batches of new data—subsets of tables, rather than whole tables. DataContexts provide simple, universal syntax to generate, fetch, and validate Batches of data from any of your DataSources.

Compile to Docs

As of v0.7.0, Great Expectations includes new classes and methods to *render* Expectations to clean, human-readable documentation. Since docs are compiled from tests and you are running tests against new data as it arrives, your documentation is guaranteed to never go stale.

1.4 Workflow advantages

Most data science and data engineering teams end up building some form of pipeline testing, eventually. Unfortunately, many teams don't get around to it until late in the game, long after early lessons from data exploration and model development have been forgotten.

In the meantime, data pipelines often become deep stacks of unverified assumptions. Mysterious (and sometimes embarrassing) bugs crop up more and more frequently. Resolving them requires painstaking exploration of upstream data, often leading to frustrating negotiations about data specs across teams.

It's not unusual to see data teams grind to a halt for weeks (or even months!) to pay down accumulated pipeline debt. This work is never fun—after all, it's just data cleaning: no new products shipped; no new insights kindled. Even

worse, it's re-cleaning old data that you thought you'd already dealt with. In our experience, servicing pipeline debt is one of the biggest productivity and morale killers on data teams.

We strongly believe that most of this pain is avoidable. We built Great Expectations to make it very, very simple to

1. set up and deploy your testing and documentation framework,
2. author Expectations through a combination of automated profiling and expert knowledge capture, and
3. systematically validate new data against them.

It's the best tool we know of for managing the complexity that inevitably grows within data pipelines. We hope it helps you as much as it's helped us.

Good night and good luck!

1.5 What does Great Expectations NOT do?

Great Expectations is NOT a pipeline execution framework.

> We aim to integrate seamlessly with DAG execution tools like Spark, Airflow, dbt, Prefect, Dagster, Kedro, etc. We DON'T execute your pipelines for you.

Great Expectations is NOT a data versioning tool.

> Great Expectations does not store data itself. Instead, it deals in metadata about data: Expectations, validation results, etc. If you want to bring your data itself under version control, check out tools like: DVC and Quilt.

Great Expectations currently works best in a Python/Bash environment.

> Great Expectations is Python-based. You can invoke it from the command line without using a Python programming environment, but if you're working in another ecosystem, other tools might be a better choice. If you're running in a pure R environment, you might consider assertR as an alternative. Within the Tensorflow ecosystem, TFDV fulfills a similar function as Great Expectations.

1.6 Who maintains Great Expectations?

Great Expectations is under active development by James Campbell, Abe Gong, Eugene Mandel and Rob Lim, with help from many others.

If you have questions, comments, or just want to have a good old-fashioned chat about data pipelines, please hop on our public Slack channel: https://greatexpectations.io/slack

If you'd like to contribute to Great Expectations, please head to the *Community* section.

If you'd like hands-on assistance setting up Great Expectations, establishing a healthy practice of data testing, or adding functionality to Great Expectations, please see options for consulting help here.

CHAPTER

TWO

GETTING STARTED

It's easy! Just use pip install:

```
$ pip install great_expectations
```

(We recommend deploying within a virtual environment. If you're not familiar with pip, virtual environments, notebooks, or git, you may want to check out the *Supporting Resources* section before continuing.)

Once Great Expectations is installed, follow this tutorial for a quick start.

2.1 Run `great_expectations init`

The *command line interface (CLI)* provides the easiest way to start using Great Expectations.

The *init* command will walk you through setting up a new project and connecting to your data.

Make sure that the machine that you installed GE on has access to a filesystem with data files (e.g., CSV) or a database.

If you prefer to use some sample data first, we suggest this example data from the United States Centers for Medicare and Medicaid Services National Provider Identifier Standard (NPI). Some later Great Expectations tutorials use this dataset, so this will make it easy to follow along.

To download this sample dataset:

```
git clone https://github.com/superconductive/ge_example_project.git
cd ge_example_project
```

Once you have decided which data you will use, you are ready to start. Run this command in the terminal:

```
great_expectations init
```

After you complete the *init* command, read this article to get a more complete picture of how data teams use Great Expectations: typical workflow.

2.2 Typical Workflow

This article describes how data teams typically use Great Expectations.

The objective of this workflow is to gain control and confidence in your data pipeline and to address the challenges of validating and monitoring the quality and accuracy of your data.

Once the setup is complete, the workflow looks like a loop over the following steps:

1. Data team members capture and document their shared understanding of their data as expectations.
2. As new data arrives in the pipeline, Great Expectations evaluates it against these expectations.
3. If the observed properties of the data are found to be different from the expected ones, the team responds by rejecting (or fixing) the data, updating the expectations, or both.

The article focuses on the "What" and the "Why" of each step in this workflow, and touches on the "How" only briefly. The exact details of configuring and executing these steps are intentionally left out - they can be found in the tutorials and reference linked from each section.

If you have not installed Great Expectations and executed the *command line interface (CLI)* init command, as described in this *tutorial*, we recommend you do so before reading the rest of the article. This will make a lot of concepts mentioned below more familiar to you.

2.2.1 Setting Up a Project

To use Great Expectations in a new data project, a *Data Context* needs to be initialized. You will see references to the Data Context throughout the documentation. A Data Context provides the core services used in a Great Expectations project.

The *CLI* command `init` does the initialization. Run this command in the terminal in the root of your project's repo:

```
great_expectations init
```

This command has to be run only once per project.

The command creates `great_expectations` subdirectory in the current directory. The team member who runs it, commits the generated directory into the version control. The contents of `great_expectations` look like this:

```
great_expectations
...
├── expectations
...
├── great_expectations.yml
├── notebooks
...
├── .gitignore
└── uncommitted
    ├── config_variables.yml
    ├── documentation
    │   └── local_site
    └── validations
```

- The `great_expectations/great_expectations.yml` configuration file defines how to access the project's data, expectations, validation results, etc.
- The `expectations` directory is where the expectations are stored as JSON files.

- The `uncommitted` directory is the home for files that should not make it into the version control - it is configured to be excluded in the `.gitignore` file. Each team member will have their own content of this directory. In Great Expectations, files should not go into the version control for two main reasons:
  - They contain sensitive information. For example, to allow the `great_expectations/great_expectations.yml` configuration file, it must not contain any database credentials and other secrets. These secrets are stored in the `uncommitted/config_variables.yml` that is not checked in.
  - They are not a "primary source of truth" and can be regenerated. For example, `uncommitted/documentation` contains generated data documentation (this article will cover data documentation in a later section).

2.2.2 Adding Datasources

Evaluating an expectation against a batch of data is the fundamental operation in Great Expectations.

For example, imagine that we have a movie ratings table in the database. This expectation says that we expect that column "rating" takes only 1, 2, 3, 4 and 5:

```
{
  "kwargs": {
    "column": "rating",
    "value_set": [1, 2, 3, 4, 5]
  },
  "expectation_type": "expect_column_distinct_values_to_be_in_set"
}
```

When Great Expectations evaluates this expectation against a dataset that has a column named "rating", it returns a validation result saying whether the data meets the expectation.

A *Datasource* is a connection to a compute environment (a backend such as Pandas, Spark, or a SQL-compatible database) and one or more storage environments.

You can have multiple Datasources in a project (Data Context). For example, this is useful if the team's pipeline consists of both a Spark cluster and a Redshift database.

All the Datasources that your project uses are configured in the project's configuration file `great_expectations/great_expectations.yml`:

```
datasources:

  our_product_postgres_database:
    class_name: SqlAlchemyDatasource
    data_asset_type:
      class_name: SqlAlchemyDataset
    credentials: ${prod_db_credentials}

  our_redshift_warehouse:
    class_name: SqlAlchemyDatasource
    data_asset_type:
      class_name: SqlAlchemyDataset
    credentials: ${warehouse_credentials}
```

The easiest way to add a datasource to the project is to use the *CLI* convenience command:

```
great_expectations datasource new
```

This command asks for the required connection attributes and tests the connection to the new Datasource.

The intrepid can add Datasources by editing the configuration file, however there are less guardrails around this approach.

A Datasource knows how to load data into the computation environment. For example, you can use a PySpark Datasource object to load data into a DataFrame from a directory on AWS S3. This is beyond the scope of this article.

After a team member adds a new Datasource to the Data Context, they commit the updated configuration file into the version control in order to make the change available to the rest of the team.

Because `great_expectations/great_expectations.yml` is committed into version control, the *CLI* command **does not store the credentials in this file**. Instead it saves them in a separate file: `uncommitted/config_variables.yml` which is not committed into version control.

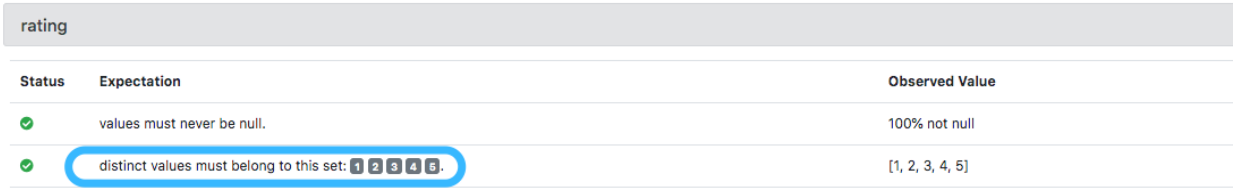
This means that that when another team member checks out the updated configuration file with the newly added Datasource, they must add their own credentials to their `uncommitted/config_variables.yml` or in environment variables.

2.2.3 Setting Up Data Docs

Data Docs is a feature of Great Expectations that creates data documentation by compiling expectations and validation results into HTML.

Data Docs produces a visual data quality report of what you expect from your data, and how the observed properties of your data differ from your expectations. It helps to keep your entire team on the same page as data evolves.

Here is what the `expect_column_distinct_values_to_be_in_set` expectation about the *rating* column of the movie ratings table from the earlier example looks like in Data Docs:



This approach to data documentation has two significant advantages.

1. **Your docs are your tests** and **your tests are your docs.** For engineers, Data Docs makes it possible to **automatically keep your data documentation in sync with your tests**. This prevents documentation rot and can save a huge amount of time and pain maintaining documentation.

2. The ability to translate expectations back and forth between human and machine-readable formats opens up many opportunities for domain experts and stakeholders who aren't engineers to collaborate more closely with engineers on data applications.

Multiple sites can be configured inside a project, each suitable for a particular use case. For example, some data teams use one site that has expectations and validation results from all the runs of their data pipeline for monitoring the pipeline's health, and another site that has only the expectations for communicating with their downstream clients. This is analogous to API documentation in software development.

To set up Data Docs for a project, an entry `data_docs_sites` must be defined in the project's configuration file. By default Data Docs site files are published to the local filesystem here: `great_expectations/uncommitted/data_docs/`. You can see this by running:

```
great_expectations docs build
```

To make a site available more broadly, a team member could configure Great Expectations to publish the site to a shared location, such as a *AWS S3*, GCS.

The site's configuration defines what to compile and where to store results. Data Docs is very customizable - see the *Data Docs Reference* for more information.

2.2.4 Authoring Expectation Suites

Earlier in this article we said that capturing and documenting the team's shared understanding of its data as expectations is the core part of this typical workflow.

Expectation Suites combine multiple expectations into an overall description of a dataset. For example, a team can group all the expectations about its `rating` table in the movie ratings database from our previous example into an Expectation Suite and call it `movieratings.ratings`. Note these names are completely flexible and the only constraint on the name of a suite is that it must be unique to a given project.

Each Expectation Suite is saved as a JSON file in the `great_expectations/expectations` subdirectory of the Data Context. Users check these files into the version control each time they are updated, same way they treat their source files. This discipline allows data quality to be an integral part of versioned pipeline releases.

The lifecycle of an Expectation Suite starts with creating it. Then it goes through an iterative loop of Review and Edit as the team's understanding of the data described by the suite evolves.

Create

While you could hand-author an Expectation Suite by writing a JSON file, just like with other features it is easier to let *CLI* save you time and typos. Run this command in the root directory of your project (where the init command created the `great_expectations` subdirectory:

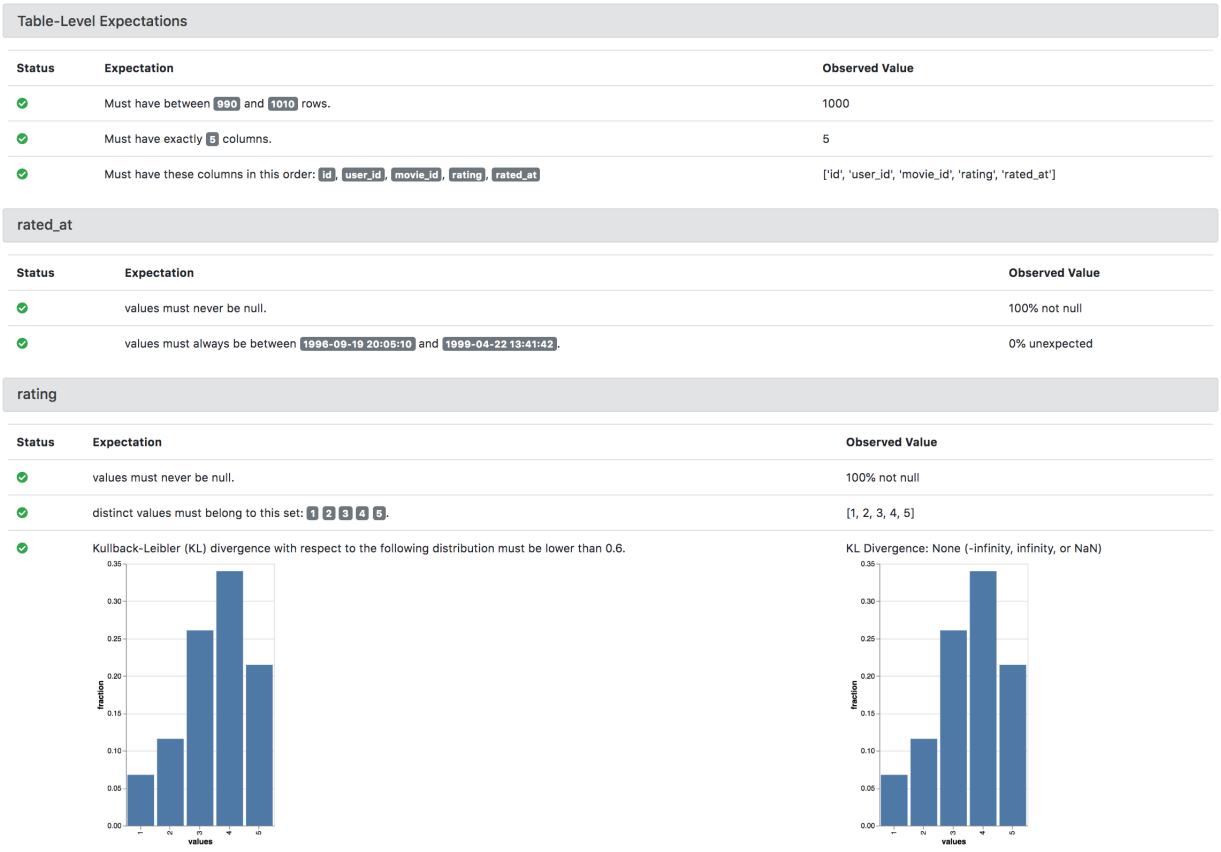
```
great_expectations suite new
```

This command prompts you to name your new Expectation Suite and to select a sample batch of data the suite will describe. Then it uses a sample of the selected data to add some initial expectations to the suite. The purpose of these is expectations is to provide examples of data assertions, and not to be meaningful. They are intended only a starting point for you to build upon.

The command concludes by saving the newly generated Expectation Suite as a JSON file and rendering the expectation suite into an HTML page in Data Docs.

Review

Reviewing expectations is best done visually in Data Docs. Here's an example of what that might look like:



Note that many of these expectations might have meaningless ranges. Also note that all expectations will have passed, since this is an example suite only. When you interactively edit your suite you will likely see failures as you iterate.

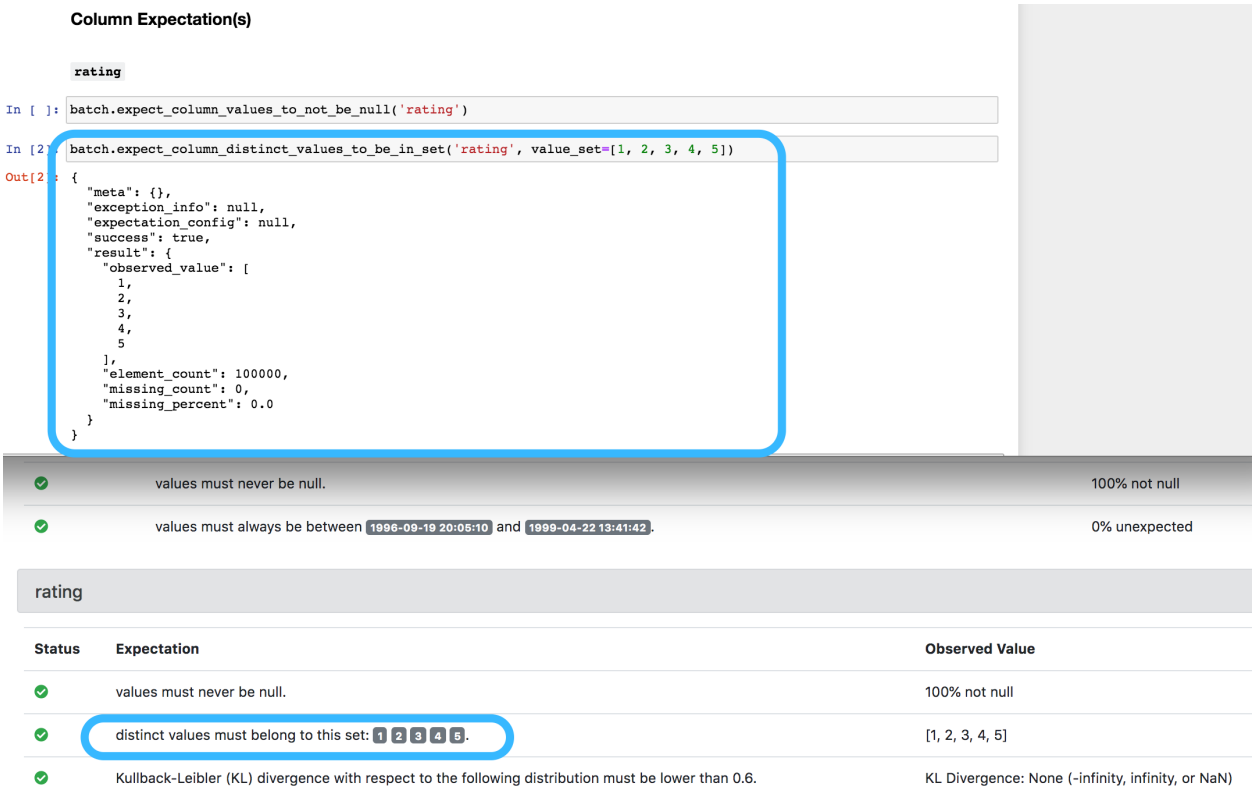
Edit

Editing an Expectation Suite means adding, removing, and modifying the arguments of existing expectations.

Similar to writing SQL queries, Expectations are best edited interactively against your data. The best interface for this is in a Jupyter notebook where you can get instant feedback as you iterate.

For every expectation type there is a Python method that sets its arguments, evaluates this expectation against a sample batch of data and adds it to the Expectation Suite.

The screenshot below shows the Python method and the Data Docs view for the same expectation (`expect_column_distinct_values_to_be_in_set`):



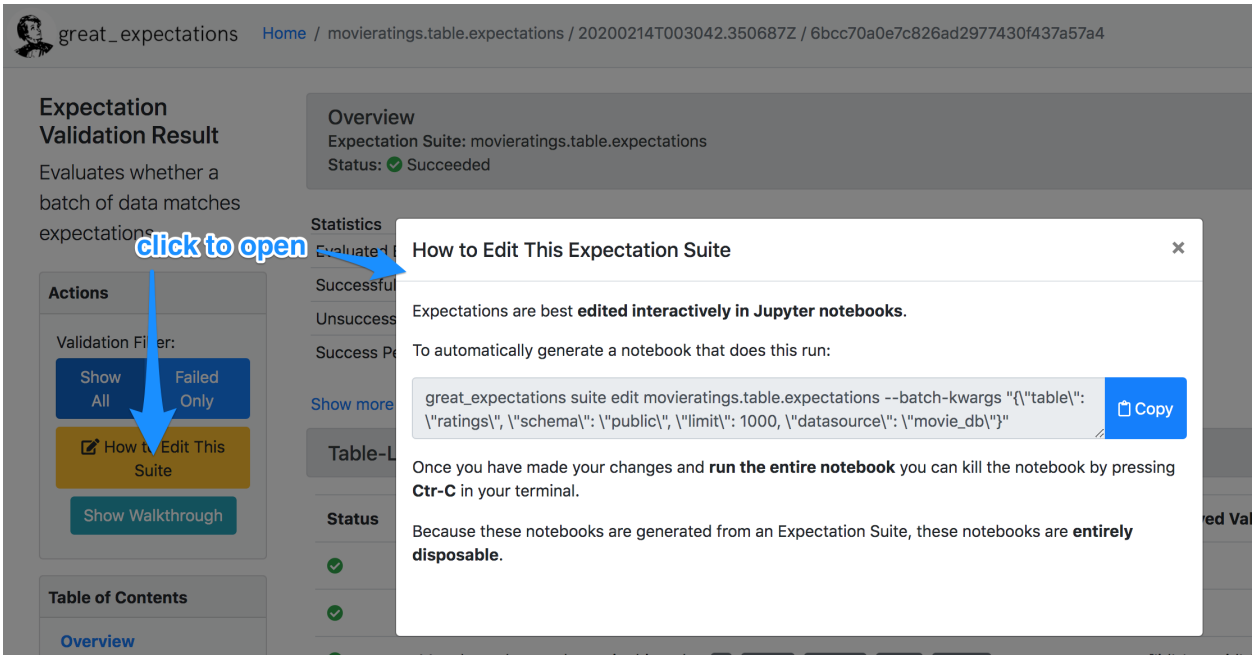
The Great Expectations *CLI* command `suite edit` generates a Jupyter notebook to edit a suite. This command saves you time by generating boilerplate that loads a batch of data and builds a cell for every expectation in the suite. This makes editing suites a breeze.

For example, to edit a suite called `movieratings.ratings` you would run:

```
great_expectations suite edit movieratings.ratings
```

These generated Jupyter notebooks can be discarded and should not be kept in source control since they are auto-generated at will, and may contain snippets of actual data.

To make this easier still, the Data Docs page for each Expectation Suite has the *CLI* command syntax for you. Simply press the "How to Edit This Suite" button, and copy/paste the *CLI* command into your terminal.



2.2.5 Validating Data using Great Expectations

So far, your team members used Great Expectations to capture and document their expectations about your data.

It is time for your team to benefit from Great Expectations' automated testing that systematically surfaces errors, discrepancies and surprises lurking in your data, allowing you and your team to be more proactive when data changes.

We typically see two main deployment patterns that we will explore in depth below.

1. Great Expectations is **deployed adjacent to your existing data pipeline**.
2. Great Expectations is **embedded into your existing data pipeline**.

Deploying automated testing adjacent to a data pipeline

You might find yourself in a situation where you do not have the engineering resources, skills, desire, or permissions to embed Great Expectations into your pipeline. As long as your data is accessible you can still reap the benefits of automated data testing.

Note: This is a fast and convenient way to get the benefits of automated data testing without requiring engineering efforts to build Great Expectations into your pipelines.

A tap is an executable python file that runs validates a batch of data against an expectation suite. Taps are a convenient way to generate a data validation script that can be run manually or via a scheduler.

Let's make a new tap using the `tap new` command.

To do this we'll specify the name of the suite and the name of the new python file we want to create. For this example, let's say we want to validate a batch of data against the `movieratings.ratings` expectation suite, and we want to make a new file called `movieratings.ratings_tap.py`

```
$ great_expectations tap new movieratings.ratings movieratings.ratings_tap.py
This is a BETA feature which may change.

Which table would you like to use? (Choose one)
    1. ratings (table)
    Don\'t see the table in the list above? Just type the SQL query
: 1
A new tap has been generated!
To run this tap, run: python movieratings.ratings_tap.py
You can edit this script or place this code snippet in your pipeline.
```

If you open the generated tap file you'll see it's only a few lines of code to get validations running! It will look like this:

```
"""
A basic generated Great Expectations tap that validates a single batch of data.

Data that is validated is controlled by BatchKwargs, which can be adjusted in
this script.

Data are validated by use of the `ActionListValidationOperator` which is
configured by default. The default configuration of this Validation Operator
saves validation results to your results store and then updates Data Docs.

This makes viewing validation results easy for you and your team.

Usage:
- Run this file: `python movieratings.ratings_tap.py`.
- This can be run manually or via a scheduler such as cron.
- If your pipeline runner supports python snippets you can paste this into your
pipeline.
"""
import sys
import great_expectations as ge

# tap configuration
context = ge.DataContext()
suite = context.get_expectation_suite("movieratings.ratings_tap")
# You can modify your BatchKwargs to select different data
batch_kwargs = {
    "table": "ratings",
    "schema": "movieratings",
    "datasource": "movieratings",
}

# tap validation process
batch = context.get_batch(batch_kwargs, suite)
results = context.run_validation_operator("action_list_operator", [batch])

if not results["success"]:
    print("Validation Failed!")
    sys.exit(1)

print("Validation Succeeded!")
sys.exit(0)
```

To run this and validate a batch of data, run:

```
$ python movieratings.ratings_tap.py
Validation Succeeded!
```

This can easily be run manually anytime you want to check your data. It can also easily be run on a schedule basis with a scheduler such as cron.

You'll want to view the detailed data quality reports in Data Docs by running `great_expectations docs build`.

For example, if you wanted to run this script nightly at 04:00, you'd add something like this to your crontab.

```
$ crontab -e
0 4 * * * /full/path/to/python /full/path/to/movieratings.ratings_tap.py
```

If you don't have access to a scheduler, you can always make checking your data part of your daily routine. Once you experience how much time and pain this saves you, we recommend geting engineering resources to embed Great Expectations validations into your pipeline.

Embedding automated testing into a data pipeline

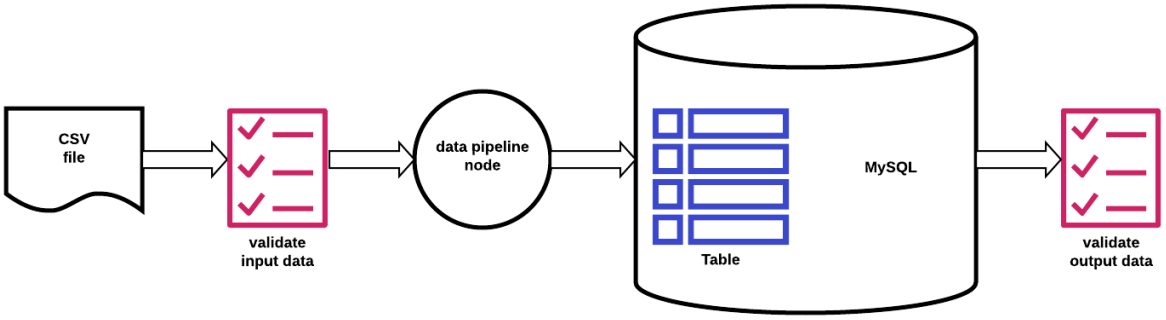
Note: This is an ideal way to deploy automated data testing if you want to take automated interventions based on the results of data validation. For example, you may want your pipeline to quarantine data that does not meet your expectations.

A data engineer can add a *Validation Operator* to your pipeline and configure it. These *Validation Operators* evaluate the new batches of data that flow through your pipeline against the expectations your team defined in the previous sections.

While data pipelines can be implemented with various technologies, at their core they are all DAGs (directed acyclic graphs) of computations and transformations over data.

This drawing shows an example of a node in a pipeline that loads data from a CSV file into a database table.

- Two expectation suites are deployed to monitor data quality in this pipeline.
- The first suite validates the pipeline's input - the CSV file - before the pipeline executes.
- The second suite validates the pipeline's output - the data loaded into the table.



To implement this validation logic, a data engineer inserts a Python code snippet into the pipeline - before and after the node. The code snippet prepares the data for the GE Validation Operator and calls the operator to perform the validation.

The exact mechanism of deploying this code snippet depends on the technology used for the pipeline.

If Airflow drives the pipeline, the engineer adds a new node in the Airflow DAG. This node will run a PythonOperator that executes this snippet. If the data is invalid, the Airflow PythonOperator will raise an error which will stop the rest of the execution.

If the pipeline uses something other than Airflow for orchestration, as long as it is possible to add a Python code snippet before and/or after a node, this will work.

Below is an example of this code snippet, with comments that explain what each line does.

```
# Data Context is a GE object that represents your project.
# Your project's great_expectations.yml contains all the config
# options for the project's GE Data Context.
context = ge.data_context.DataContext()

datasource_name = "my_production_postgres" # a datasource configured in your great_
↪expectations.yml

# Tell GE how to fetch the batch of data that should be validated...

# ... from the result set of a SQL query:
batch_kwargs = {"query": "your SQL query", "datasource": datasource_name}

# ... or from a database table:
# batch_kwargs = {"table": "name of your db table", "datasource": datasource_name}

# ... or from a file:
# batch_kwargs = {"path": "path to your data file", "datasource": datasource_name}

# ... or from a Pandas or PySpark DataFrame
# batch_kwargs = {"dataset": "your Pandas or PySpark DataFrame", "datasource":␣
↪datasource_name}

# Get the batch of data you want to validate.
# Specify the name of the expectation suite that holds the expectations.
expectation_suite_name = "movieratings.ratings" # this is an example of
                                                # a suite that you created
batch = context.get_batch(batch_kwargs, expectation_suite_name)

# Call a validation operator to validate the batch.
# The operator will evaluate the data against the expectations
# and perform a list of actions, such as saving the validation
# result, updating Data Docs, and firing a notification (e.g., Slack).
results = context.run_validation_operator(
    "action_list_operator",
    assets_to_validate=[batch],
    run_id=run_id) # e.g., Airflow run id or some run identifier that your pipeline␣
↪uses.

if not results["success"]:
    # Decide what your pipeline should do in case the data does not
    # meet your expectations.
```

2.2.6 Responding to Validation Results

A *Validation Operator* is deployed at a particular point in your data pipeline.

A new batch of data arrives and the operator validates it against an expectation suite (see the previous step).

The *actions* of the operator store the validation result, add an HTML view of the result to the Data Docs website, and fire a configurable notification (by default, Slack).

If the data meets all the expectations in the suite, no action is required. This is the beauty of automated testing. No team members have to be interrupted.

In case the data violates some expectations, team members must get involved.

In the world of software testing, if a program does not pass a test, it usually means that the program is wrong and must be fixed.

In pipeline and data testing, if data does not meet expectations, the response to a failing test is triaged into 3 categories:

1. **The data is fine, and the validation result revealed a characteristic that the team was not aware of.**

   The team's data scientists or domain experts update the expectations to reflect this new discovery. They use the process described above in the Review and Edit sections to update the expectations while testing them against the data batch that failed validation.

2. **The data is "broken"**, and **can be recovered.**

   For example, the users table could have dates in an incorrect format. Data engineers update the pipeline code to deal with this brokenness and fix it on the fly.

3. **The data is "broken beyond repair".**

   The owners of the pipeline go upstream to the team (or external partner) who produced the data and address it with them. For example, columns in the users table could be missing entirely. The validation results in Data Docs makes it easy to communicate exactly what is broken, since it shows the expectation that was not met and observed examples of non-conforming data.

CHAPTER

THREE

GLOSSARY OF EXPECTATIONS

This is a list of all built-in Expectations. Expectations are extendable so you can create custom expectations for your data domain! To do so see this article: *Building Custom Expectations*.

3.1 Dataset

Dataset objects model tabular data and include expectations with row and column semantics. Many Dataset expectations are implemented using column_map_expectation and column_aggregate_expectation decorators.

Not all expectations are currently implemented for each backend. Please see *Table of Expectation Implementations By Backend*.

3.1.1 Table shape

- *expect_column_to_exist*
- *expect_table_columns_to_match_ordered_list*
- *expect_table_row_count_to_be_between*
- *expect_table_row_count_to_equal*

3.1.2 Missing values, unique values, and types

- *expect_column_values_to_be_unique*
- *expect_column_values_to_not_be_null*
- *expect_column_values_to_be_null*
- *expect_column_values_to_be_of_type*
- *expect_column_values_to_be_in_type_list*

3.1.3 Sets and ranges

- *expect_column_values_to_be_in_set*
- *expect_column_values_to_not_be_in_set*
- *expect_column_values_to_be_between*
- *expect_column_values_to_be_increasing*
- *expect_column_values_to_be_decreasing*

3.1.4 String matching

- *expect_column_value_lengths_to_be_between*
- *expect_column_value_lengths_to_equal*
- *expect_column_values_to_match_regex*
- *expect_column_values_to_not_match_regex*
- *expect_column_values_to_match_regex_list*
- *expect_column_values_to_not_match_regex_list*

3.1.5 Datetime and JSON parsing

- *expect_column_values_to_match_strftime_format*
- *expect_column_values_to_be_dateutil_parseable*
- *expect_column_values_to_be_json_parseable*
- *expect_column_values_to_match_json_schema*

3.1.6 Aggregate functions

- *expect_column_distinct_values_to_be_in_set*
- *expect_column_distinct_values_to_contain_set*
- *expect_column_distinct_values_to_equal_set*
- *expect_column_mean_to_be_between*
- *expect_column_median_to_be_between*
- *expect_column_quantile_values_to_be_between*
- *expect_column_stdev_to_be_between*
- *expect_column_unique_value_count_to_be_between*
- *expect_column_proportion_of_unique_values_to_be_between*
- *expect_column_most_common_value_to_be_in_set*
- *expect_column_max_to_be_between*
- *expect_column_min_to_be_between*
- *expect_column_sum_to_be_between*

3.1.7 Multi-column

- *expect_column_pair_values_A_to_be_greater_than_B*
- *expect_column_pair_values_to_be_equal*
- *expect_column_pair_values_to_be_in_set*
- *expect_multicolumn_values_to_be_unique*

3.1.8 Distributional functions

- *expect_column_kl_divergence_to_be_less_than*
- *expect_column_bootstrapped_ks_test_p_value_to_be_greater_than*
- *expect_column_chisquare_test_p_value_to_be_greater_than*
- *expect_column_parameterized_distribution_ks_test_p_value_to_be_greater_than*

3.2 FileDataAsset

File data assets reason at the file level, and the line level (for text data).

- *expect_file_line_regex_match_count_to_be_between*
- *expect_file_line_regex_match_count_to_equal*
- *expect_file_hash_to_equal*
- *expect_file_size_to_be_between*
- *expect_file_to_exist*
- *expect_file_to_have_valid_table_header*
- *expect_file_to_be_valid_json*

CHAPTER

FOUR

THE GREAT EXPECTATIONS COMMAND LINE

After reading this guide, you will know:

- How to create a Great Expectations project
- How to add new datasources
- How to add and edit expectation suites
- How to build and open Data Docs

The Great Expectations command line is organized using a **<NOUN> <VERB>** syntax. This guide is organized by nouns (datasource, suite, docs) then verbs (new, list, edit, etc).

4.1 Basics

There are a few commands that are critical to your everyday usage of Great Expectations. This is a list of the most common commands you'll use in order of how much you'll probably use them:

- `great_expectations suite edit`
- `great_expectations suite new`
- `great_expectations suite list`
- `great_expectations docs build`
- `great_expectations tap new`
- `great_expectations validation-operator run`
- `great_expectations datasource list`
- `great_expectations datasource new`
- `great_expectations datasource profile`
- `great_expectations init`

You can get a list of Great Expectations commands available to you by typing `great_expectations --help`. Each noun command and each verb sub-command has a description, and should help you find the thing you need.

Note: All Great Expectations commands have help text. As with most posix utilities, you can try adding `--help` to the end. For example, by running `great_expectations suite new --help` you'll see help output for that specific command.

```
$ great_expectations --help
Usage: great_expectations [OPTIONS] COMMAND [ARGS]...

  Welcome to the great_expectations CLI!

  Most commands follow this format: great_expectations <NOUN> <VERB>
  The nouns are: datasource, docs, project, suite
  Most nouns accept the following verbs: new, list, edit

  In particular, the CLI supports the following special commands:

  - great_expectations init : create a new great_expectations project
  - great_expectations datasource profile : profile a  datasource
  - great_expectations docs build : compile documentation from expectations

Options:
  --version      Show the version and exit.
  -v, --verbose  Set great_expectations to use verbose output.
  --help         Show this message and exit.

Commands:
  datasource  datasource operations
  docs        data docs operations
  init        initialize a new Great Expectations project
  project     project operations
  suite       expectation suite operations
```

4.2 great_expectations init

To add Great Expectations to your project run the `great_expectations init` command in your project directory. This will run you through a very short interactive experience to connect to your data, show you some sample expectations, and open Data Docs.

Note: You can install the Great Expectations python package by typing `pip install great_expectations`, if you don't have it already.

```
$ great_expectations init
  ...
```

After this command has completed, you will have the entire Great Expectations directory structure with all the code you need to get started protecting your pipelines and data.

4.3 great_expectations docs

4.3.1 great_expectations docs build

The `great_expectations docs build` command builds your Data Docs site. You'll use this any time you want to view your expectations and validations in a web browser.

```
$ great_expectations docs build
Building Data Docs...
The following Data Docs sites were built:
- local_site:
   file:///Users/dickens/my_pipeline/great_expectations/uncommitted/data_docs/local_
↪site/index.html
```

4.4 great_expectations suite

All command line operations for working with expectation suites are here.

4.4.1 great_expectations suite list

Running `great_expectations suite list` gives a list of available expectation suites in your project:

```
$ great_expectations suite list
3 expectation suites found:
    customer_requests.warning
    customer_requests.critical
    churn_model_input
```

4.4.2 great_expectations suite new

Create a new expectation suite. Just as writing SQL queries is far better with access to data, so are writing expectations. These are best written interactively against some data.

To this end, this command interactively helps you choose some data, creates the new suite, adds sample expectations to it, and opens up Data Docs.

Important: The sample suites generated **are not meant to be production suites** - they are examples only.

Great Expectations will choose a couple of columns and generate expectations about them to demonstrate some examples of assertions you can make about your data.

```
$ great_expectations suite new
Enter the path (relative or absolute) of a data file
: data/npi.csv

Name the new expectation suite [npi.warning]:

Great Expectations will choose a couple of columns and generate expectations about␣
↪them
to demonstrate some examples of assertions you can make about your data.
```

(continues on next page)

(continued from previous page)

```
Press Enter to continue
:

Generating example Expectation Suite...
Building Data Docs...
The following Data Docs sites were built:
- local_site:
   file:///Users/dickens/Desktop/great_expectations/uncommitted/data_docs/local_site/
↪index.html
A new Expectation suite 'npi.warning' was added to your project
```

To edit this suite you can click the **How to edit** button in Data Docs, or run the command: `great_expectations suite edit npi.warning`. This will generate a jupyter notebook and allow you to add, remove or adjust any expectations in the sample suite.

Important: Great Expectations generates working jupyter notebooks when you make new suites and edit existing ones. This saves you tons of time by avoiding all the necessary boilerplate.

Because these notebooks can be generated at any time from the expectation suites (stored as JSON) you should **consider the notebooks to be entirely disposable artifacts**.

They are put in your `great_expectations/uncommitted` directory and you can delete them at any time.

Because they can expose actual data, we strongly suggest leaving them in the `uncommitted` directory to avoid potential data leaks into source control.

4.4.3 great_expectations suite new --suite <SUITE_NAME>

If you already know the name of the suite you want to create you can skip one of the interactive prompts and specify the suite name directly.

```
$ great_expectations suite new --suite npi.warning
Enter the path (relative or absolute) of a data file
: data/npi.csv
... (same as above)
```

4.4.4 great_expectations suite new --empty

If you prefer to skip the example expectations and start writing expectations in a new empty suite directly in a jupyter notebook, add the `--empty` flag.

```
$ great_expectations suite new --empty
Enter the path (relative or absolute) of a data file
: data/npi.csv

Name the new expectation suite [npi.warning]: npi.warning
A new Expectation suite 'npi.warning' was added to your project
Because you requested an empty suite, we'll open a notebook for you now to edit it!
If you wish to avoid this you can add the `--no-jupyter` flag.
```

(continues on next page)

(continued from previous page)

```
[I 14:55:15.992 NotebookApp] Serving notebooks from local directory: /Users/dickens/
↪Desktop/great_expectations/uncommitted
... (jupyter opens)
```

4.4.5 great_expectations suite new --empty --no-jupyter

If you prefer to disable Great Expectations from automatically opening the generated jupyter notebook, add the `--no-jupyter` flag.

```
$ great_expectations suite new --empty --no-jupyter

Enter the path (relative or absolute) of a data file
: data/npi.csv

Name the new expectation suite [npi.warning]: npi.warning
A new Expectation suite 'npi.warning' was added to your project
To continue editing this suite, run jupyter notebook /Users/taylor/Desktop/great_
↪expectations/uncommitted/npi.warning.ipynb
```

You can then run jupyter.

4.4.6 great_expectations suite edit

Edit an existing expectation suite. Just as writing SQL queries is far better with access to data, so are authoring expectations. These are best authored interactively against some data. This best done in a jupyter notebook.

Note: BatchKwargs define what data to use during editing.

- When suites are created through the CLI, the original batch_kwargs are stored in a piece of metadata called a citation.
- The edit command uses the most recent batch_kwargs as a way to know what data should be used for the interactive editing experience.
- It is often desirable to edit the suite on a different chunk of data.
- To do this you can edit the batch_kwargs in the generated notebook.

To this end, this command interactively helps you choose some data, generates a working jupyter notebook, and opens up that notebook in jupyter.

```
$ great_expectations suite edit npi.warning
[I 15:22:18.809 NotebookApp] Serving notebooks from local directory: /Users/dickens/
↪Desktop/great_expectations/uncommitted
... (juypter runs)
```

Important: Great Expectations generates working jupyter notebooks when you make new suites and edit existing ones. This saves you tons of time by avoiding all the necessary boilerplate.

Because these notebooks can be generated at any time from the expectation suites (stored as JSON) you should **consider the notebooks to be entirely disposable artifacts**.

They are put in your `great_expectations/uncommitted` directory and you can delete them at any time.

Because they can expose actual data, we strongly suggest leaving them in the `uncommitted` directory to avoid potential data leaks into source control.

4.4.7 great_expectations suite edit <SUITE_NAME> --no-jupyter

If you prefer to disable Great Expectations from automatically opening the generated jupyter notebook, add the `--no-jupyter` flag.

```
$ great_expectations suite edit npi.warning --no-jupyter
To continue editing this suite, run jupyter notebook /Users/dickens/Desktop/great_
↪expectations/uncommitted/npi.warning.ipynb
```

You can then run jupyter.

4.5 great_expectations validation-operator

All command line operations for working with *validation operators* are here.

4.5.1 great_expectations validation-operator list

Running `great_expectations validation-operator list` gives a list of validation operators configured in your project:

```
$ great_expectations validation-operator list
... (YOUR VALIDATION OPERATORS)
```

4.5.2 great_expectations validation-operator run

There are two modes to run this command:

1. Interactive (good for development):

Specify the name of the validation operator using the `--name` argument and the name of the expectation suite using the `--suite` argument.

The cli will help you specify the batch of data that you want to validate interactively.

If you want to call a validation operator to validate one batch of data against one expectation suite, you can invoke this command:

`great_expectations validation-operator run --name <VALIDATION_OPERATOR_NAME> --suite <SUITE_NAME>`

Use the *–name* argument to specify the name of the validation operator you want to run. This has to be the name of one of the validation operators configured in your project. You can list the names by calling the `great_expectations validation-operator list` command or by examining the `validation_operators` section in your project's `great_expectations.yml` config file.

Use the *–suite* argument to specify the name of the expectation suite you want the validation operator to validate the batch of data against. This has to be the name of one of the expectation suites that exist in your project. You can look up the names by calling the *suite list* command.

The command will help you specify the batch of data that you want the validation operator to validate interactively.

```
$ great_expectations validation-operator --name action_list_operator --suite npi.
↪warning

Let's help you specify the batch of data your want the validation operator to␣
↪validate.

Enter the path (relative or absolute) of a data file
: data/npi_small.csv
Validation Succeeded!
```

2. Non-interactive (good for production):

If you want run a validation operator non-interactively, use the *–validation_config_file* argument to specify the path of the validation configuration JSON file.

`great_expectations validation-operator run ----validation_config_file <VALIDATION_CONFIG_FILE_PATH>`

This file can be used to instruct a validation operator to validate multiple batches of data and use multiple expectation suites to validate each batch.

Note: A validation operator can validate multiple batches of data and use multiple expectation suites to validate each batch. For example, you might want to validate 3 source files, with 2 tiers of suites each, one for a warning suite and one for a critical stop-the-presses hard failure suite.

This command exits with 0 if the validation operator ran and the "success" attribute in its return object is True. Otherwise, the command exits with 1.

Tip: This is an excellent way to use call Great Expectations from within your pipeline if your pipeline code can run shell commands.

A validation config file specifies the name of the validation operator in your project and the list of batches of data that you want the operator to validate. Each batch is defined using `batch_kwargs`. The `expectation_suite_names` attribute for each batch specifies the list of names of expectation suites that the validation operator should use to validate the batch.

Here is an example validation config file:

```
{
  "validation_operator_name": "action_list_operator",
  "batches": [
    {
      "batch_kwargs": {
        "path": "/Users/me/projects/my_project/data/data.csv",
        "datasource": "my_filesystem_datasource",
        "reader_method": "read_csv"
      },
      "expectation_suite_names": ["suite_one", "suite_two"]
    },
    {
      "batch_kwargs": {
        "query": "SELECT * FROM users WHERE status = 1",
```

(continues on next page)

(continued from previous page)

```
        "datasource": "my_redshift_datasource"
      },
      "expectation_suite_names": ["suite_three"]
    }
  ]
}
```

```
$ great_expectations validation-operator run --validation_config_file my_val_config.
↪json
Validation Succeeded!
```

4.6 great_expectations datasource

All command line operations for working with *datasources* are here. A datasource is a connection to data and a processing engine. Examples of a datasource are: - csv files processed in pandas or Spark - a relational database such as Postgres, Redshift or BigQuery

4.6.1 great_expectations datasource list

This command displays a list of your datasources and their types. These can be found in your `great_expectations/great_expectations.yml` config file.

```
$ great_expectations datasource list
[{'name': 'files_datasource', 'class_name': 'PandasDatasource'}]
```

4.6.2 great_expectations datasource new

This interactive command helps you connect to your data.

```
$ great_expectations datasource list
What data would you like Great Expectations to connect to?
1. Files on a filesystem (for processing with Pandas or Spark)
2. Relational database (SQL)
: 1

What are you processing your files with?
    1. Pandas
    2. PySpark
: 1

Enter the path (relative or absolute) of the root directory where the data files are␣
↪stored.
: data

Give your new data source a short name.
 [data__dir]: npi_drops
A new datasource 'npi_drops' was added to your project.
```

If you are using a database you will be guided through a series of prompts that collects and verifies connection details and credentials.

4.6.3 great_expectations datasource profile

For details on profiling, see this *reference document*

Caution: Profiling is a beta feature and is not guaranteed to be stable. YMMV

4.7 great_expectations tap

All command line operations for working with taps are here. A tap is an executable python file that runs validations that you can create to aid deployment of validations.

4.7.1 great_expectations tap new

Creating a tap requires a valid suite name and tap filename. This is the name of a python file that this command will write to.

Note: Taps are a beta feature to speed up deployment. Please open a new issue if you discover a use case that does not yet work or have ideas how to make this feature better!

```
$ great_expectations tap new npi.warning npi.warning.py
This is a BETA feature which may change.

Enter the path (relative or absolute) of a data file
: data/npi.csv
A new tap has been generated!
To run this tap, run: python npi.warning.py
You can edit this script or place this code snippet in your pipeline.
```

You will now see a new tap file on your filesystem.

This can be run by invoking it with:

```
$ python  npi.warning.py
Validation Suceeded!
$ echo $?
0
```

This posix-compatible exits with a status of `0` if validation is successful and a status of `1` if validation failed.

A failure will look like:

```
$ python  npi.warning.py
Validation Failed!
$ echo $?
1
```

The *Typical Workflow* document shows you how taps can be embedded in your existing pipeline or used adjacent to a pipeline.

If you are using a SQL datasource you will be guided through a series of prompts that helps you choose a table or write a SQL query.

Tip: A custom SQL query can be very handy if for example you wanted to validate all records in a table with timestamps.

For example, imagine you have a machine learning model that looks at the last 14 days of customer events to predict churn. If you have built a suite called `churn_model_assumptions` and a postgres database with a `user_events` table with an `event_timestamp` column and you wanted to validate all events that occurred in the last 14 days you might do something like:

```
$ great_expectations tap new churn_model_assumptions churn_model_assumptions.py
This is a BETA feature which may change.

Which table would you like to use? (Choose one)
1. user_events (table)
Don't see the table in the list above? Just type the SQL query
: SELECT * FROM user_events WHERE event_timestamp > now() - interval '14 day';
A new tap has been generated!
To run this tap, run: python churn_model_assumptions.py
You can edit this script or place this code snippet in your pipeline.
```

This tap can then be run nightly before your model makes churn predictions!

4.8 Miscellaneous

- `great_expectations project check-config` checks your `great_expectations/great_expectations.yml` for validity. This is handy for occasional Great Expectations version migrations.

4.9 Acknowledgements

This article was heavily inspired by the phenomenal Rails Command Line Guide https://guides.rubyonrails.org/command_line.html.

CHAPTER

FIVE

TUTORIALS

This is a collection of tutorials that walk you through a variety of useful Great Expectations workflows.

5.1 Create Expectations

This tutorial covers the workflow of creating and editing expectations.

The tutorial assumes that you have created a new Data Context (project), as covered here: *Run great_expectations init*.

Creating expectations is an opportunity to blend contextual knowledge from subject-matter experts and insights from profiling and performing exploratory analysis on your dataset.

Once the initial setup of Great Expectations is complete, the workflow looks like a loop over the following steps:

1. Data team members capture and document their shared understanding of their data as expectations.
2. As new data arrives in the pipeline, Great Expectations evaluates it against these expectations.
3. If the observed properties of the data are found to be different from the expected ones, the team responds by rejecting (or fixing) the data, updating the expectations, or both.

For a broader understanding of the typical workflow read this article: typical_workflow.

Expectations are grouped into Expectations Suites. An Expectation Suite combines multiple expectations into an overall description of a dataset. For example, a team can group all the expectations about the `rating` table in the movie ratings database into an Expectation Suite and call it "movieratings.table.expectations".

Each Expectation Suite is saved as a JSON file in the `great_expectations/expectations` subdirectory of the Data Context. Users check these files into version control each time they are updated, in the same way they treat their source files.

The lifecycle of an Expectation Suite starts with creating it. Then it goes through a loop of Review and Edit as the team's understanding of the data described by the suite evolves.

We will describe the Create, Review and Edit steps in brief:

5.1.1 Create an Expectation Suite

Expectation Suites are saved as JSON files, so you *could* create a new suite by writing a file directly. However the preferred way is to let the CLI save you time and typos. If you cannot use the *CLI* in your environment (e.g., in a Databricks cluster), you can create and edit an Expectation Suite in a notebook. Jump to this section for details: *Jupyter Notebook for Creating and Editing Expectation Suites*.

To continue with the *CLI*, run this command in the root directory of your project (where the init command created the `great_expectations` subdirectory:

```
great_expectations suite new
```

This command prompts you to name your new Expectation Suite and to select a sample batch of the dataset the suite will describe. Then it profiles the selected sample and adds some initial expectations to the suite. The purpose of these expectations is to provide examples of what properties of data can be described using Great Expectations. They are only a starting point that the user builds on.

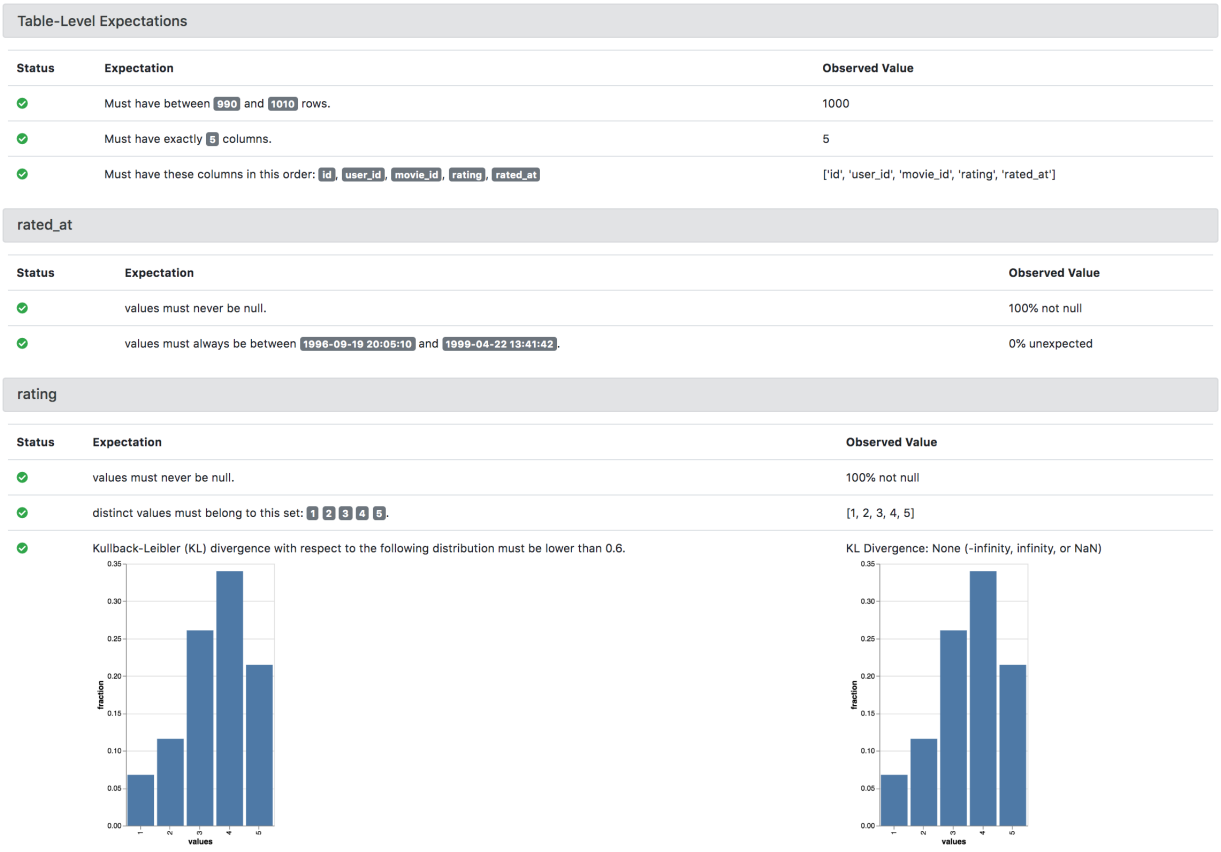
The command concludes by saving the newly generated Expectation Suite as a JSON file and rendering the expectation suite into an HTML page in the Data Docs website of the Data Context.

5.1.2 Review an Expectation Suite

Data Docs is a feature of Great Expectations that creates data documentation by compiling expectations and validation results into HTML.

Data Docs produces a visual data quality report of what you expect from your data, and how the observed properties of your data differ from your expectations. It helps to keep your entire team on the same page as data evolves.

Reviewing expectations is best done in Data Docs:



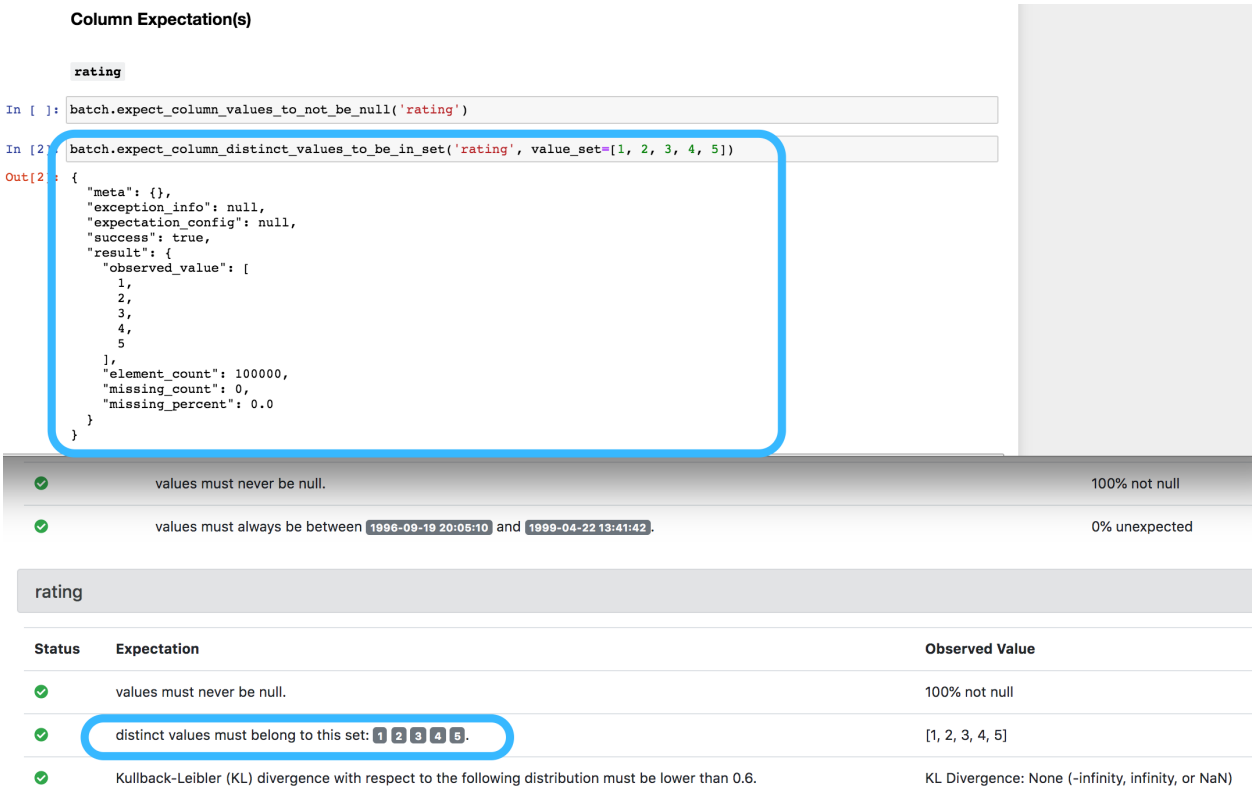
5.1.3 Edit an Expectation Suite

The best interface for editing an Expectation Suite is a Jupyter notebook.

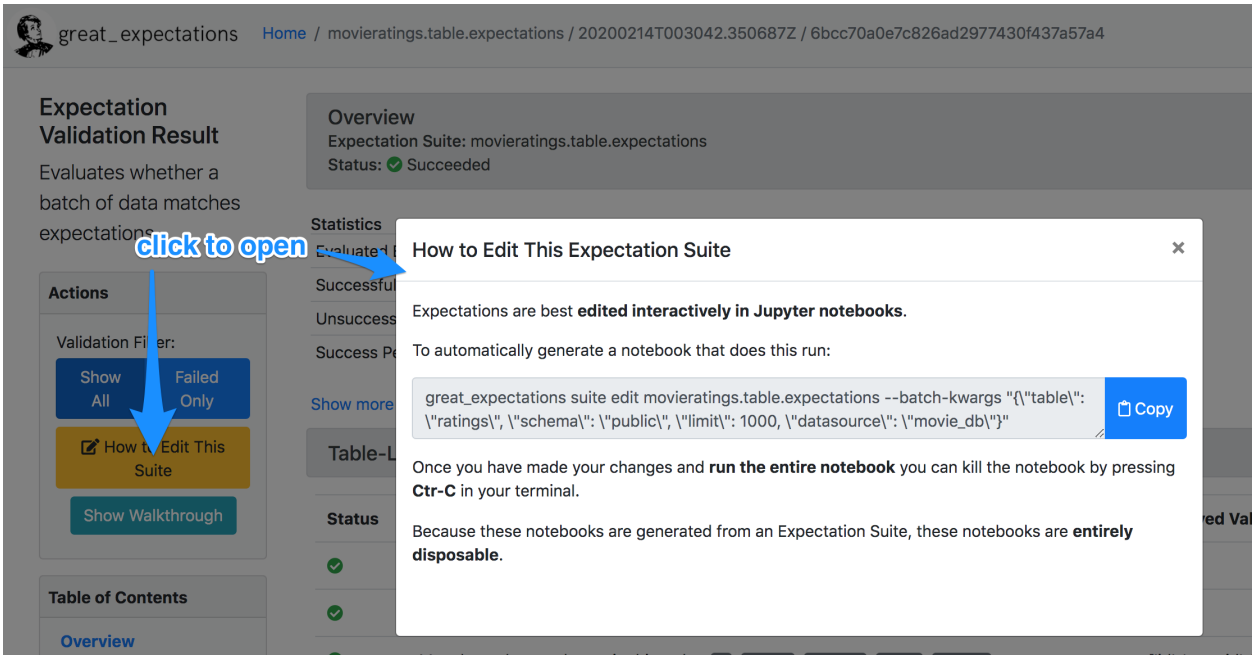
Editing an Expectation Suite means adding expectations, removing expectations, and modifying the arguments of existing expectations.

For every expectation type there is a Python method that sets its arguments, evaluates this expectation against a sample batch of data and adds it to the Expectation Suite.

Take a look at the screenshot below. It shows the HTML view and the Python method for the same expectation (`expect_column_distinct_values_to_be_in_set`) side by side:



The *CLI* provides a command that, given an Expectation Suite, generates a Jupyter notebook to edit it. It takes care of generating a cell for every expectation in the suite and of getting a sample batch of data. The HTML page for each Expectation Suite has the CLI command syntax in order to make it easier for users.



The generated Jupyter notebook can be discarded, since it is auto-generated.

To understand this auto-generated notebook in more depth, jump to this section: *Jupyter Notebook for Creating and Editing Expectation Suites*.

5.1.4 Jupyter Notebook for Creating and Editing Expectation Suites

If you used the *CLI suite new* command to create an Expectation Suite and then the *suite edit* command to edit it, then the CLI generated a notebook in the `great_expectations/uncommitted/` folder for you. There is no need to check this notebook in to version control. Next time you decide to edit this Expectation Suite, use the *CLI* again to generate a new notebook that reflects the expectations in the suite at that time.

If you do not use the *CLI*, create a new notebook in the``great_expectations/notebooks/`` folder in your project.

1. Setup

```
from datetime import datetime
import great_expectations as ge
import great_expectations.jupyter_ux
from great_expectations.data_context.types.resource_identifiers import␣
↪ValidationResultIdentifier

# Data Context is a GE object that represents your project.
# Your project's great_expectations.yml contains all the config
# options for the project's GE Data Context.
context = ge.data_context.DataContext()

# Create a new empty Expectation Suite
# and give it a name
expectation_suite_name = "ratings.table.warning" # this is just an example
context.create_expectation_suite(
expectation_suite_name)
```

If an expectation suite with this name already exists for this data_asset, you will get an error. If you would like to overwrite this expectation suite, set `overwrite_existing=True`.

2. Load a batch of data to create Expectations

Select a sample batch of the dataset the suite will describe.

`batch_kwargs` provide detailed instructions for the Datasource on how to construct a batch. Each Datasource accepts different types of `batch_kwargs` - regardless of Datasource type, a Datasource name must always be provided:

pandas

A pandas datasource can accept `batch_kwargs` that describe either a path to a file or an existing DataFrame. For example, if the data asset is a collection of CSV files in a folder that are processed with Pandas, then a batch could be one of these files. Here is how to construct `batch_kwargs` that specify a particular file to load:

```
batch_kwargs = {
    'path': "PATH_OF_THE_FILE_YOU_WANT_TO_LOAD",
    'datasource': "DATASOURCE_NAME"
}
```

To instruct `get_batch` to read CSV files with specific options (e.g., not to interpret the first line as the header or to use a specific separator), add them to the `batch_kwargs` under the "reader_options" key.

See the complete list of options for Pandas read_csv.

`batch_kwargs` might look like the following:

```
{
    "path": "/data/npidata/npidata_pfile_20190902-20190908.csv",
    "datasource": "files_datasource",
    "reader_options": {
        "sep": "|"
    }
}
```

If you already loaded the data into a Pandas DataFrame called *df*, you could use following `batch_kwargs` to instruct the datasource to use your DataFrame as a batch:

```
batch_kwargs = {
    'dataset': df,
    'datasource': 'files_datasource'
}
```

pyspark

A pyspark datasource can accept `batch_kwargs` that describe either a path to a file or an existing DataFrame. For example, if the data asset is a collection of CSV files in a folder that are processed with Pandas, then a batch could be one of these files. Here is how to construct `batch_kwargs` that specify a particular file to load:

```
batch_kwargs = {
    'path': "PATH_OF_THE_FILE_YOU_WANT_TO_LOAD",
    'datasource': "DATASOURCE_NAME"
}
```

To instruct `get_batch` to read CSV files with specific options (e.g., not to interpret the first line as the header or to use a specific separator), add them to the `batch_kwargs` under the "reader_options" key.

See the complete list of options for Spark DataFrameReader

SQLAlchemy

A SQLAlchemy datasource can accept `batch_kwargs` that instruct it load a batch from a table, a view, or a result set of a query:

If you would like to validate an entire table (or a view) in your database's default schema:

```
batch_kwargs = {
    'table': "YOUR TABLE NAME",
    'datasource': "DATASOURCE_NAME"
}
```

If you would like to validate an entire table or view from a non-default schema in your database:

```
batch_kwargs = {
    'table': "YOUR TABLE NAME",
    'schema': "YOUR SCHEMA",
    'datasource': "DATASOURCE_NAME"
}
```

If you would like to validate using a query to construct a temporary table:

```
batch_kwargs = {
    'query': 'SELECT YOUR_ROWS FROM YOUR_TABLE',
    'datasource': "DATASOURCE_NAME"
}
```

The DataContext's `get_batch` method is used to load a batch of a data asset:

```
batch = context.get_batch(batch_kwargs, expectation_suite_name)
batch.head()
```

Calling this method asks the Context to get a batch of data and attach the expectation suite `expectation_suite_name` to it. The `batch_kwargs` argument specifies which batch of the data asset should be loaded.

3. Author Expectations

Now that you have a batch of data, you can call `expect` methods on the data asset in order to check whether this expectation is true for this batch of data.

For example, to check whether it is reasonable to expect values in the column "NPI" to never be empty, call: `batch.expect_column_values_to_not_be_null('NPI')`

Some expectations can be created from your domain expertise; for example we might expect that most entries in the NPI database use the title "Dr." instead of "Ms.", or we might expect that every row should use a unique value in the 'NPI' column.

Here is how we can add an expectation that expresses that knowledge:

```
batch.expect_column_values_to_be_unique('NPI')
```

```
{'success': True,
 'result': {'element_count': 18877,
  'missing_count': 0,
  'missing_percent': 0.0,
  'unexpected_count': 0,
  'unexpected_percent': 0.0,
  'unexpected_percent_nonmissing': 0.0,
  'partial_unexpected_list': []}}
```

Other expectations can be created by examining the data in the batch. For example, suppose you want to protect a pipeline against improper values in the "Provider Other Organization Name Type Code" column. Even if you don't know exactly what the "improper" values are, you can explore the data by trying some values to check if the data in the batch meets your expectation:

```
batch.expect_column_values_to_be_in_set("Provider Other Organization Name Type Code",
                                        [0, 1, 2, 3])
```

```
{'success': False,
 'result': {'element_count': 18877,
  'missing_count': 17819,
  'missing_percent': 94.39529586269005,
  'unexpected_count': 124,
  'unexpected_percent': 0.6568840387773481,
  'unexpected_percent_nonmissing': 11.720226843100189,
  'partial_unexpected_list': [5.0,
   4.0,
   5.0,
   5.0,
   5.0,
   5.0,
   5.0,
   5.0,
   5.0,
   4.0,
   5.0,
   4.0,
   5.0,
   5.0,
   4.0,
   5.0,
   5.0,
   5.0,
   5.0]}}
```

Validating the expectation against the batch resulted in failure - there are some values in the column that do not meet the expectation. The "partial_unexpected_list" key in the result dictionary contains examples of non-conforming values. Examining these examples shows that some titles are not in the expected set. Adjust the `value_set` and rerun the expectation method:

```
batch.expect_column_values_to_be_in_set("Provider Other Organization Name Type Code",
                                        [0, 1, 2, 3, 4, 5])
```

```
{'success': True,
 'result': {'element_count': 18877,
  'missing_count': 17819,
  'missing_percent': 94.39529586269005,
  'unexpected_count': 0,
  'unexpected_percent': 0.0,
  'unexpected_percent_nonmissing': 0.0,
  'partial_unexpected_list': []}}
```

This time validation was successful - all values in the column meet the expectation.

Although you called `expect_column_values_to_be_in_set` twice (with different argument values), only one expectation of type `expect_column_values_to_be_in_set` will be created for the column - the latest call overrides all the earlier ones. By default, only expectations that were true on their last run are saved.

How do I know which types of expectations I can add?

- *Tab-complete* the partially typed `expect_` method name to see available expectations.
- In Jupyter, you can also use *shift-tab* to see the docstring for each expectation, including the parameters it takes and to get more information about the expectation.
- Visit the *Glossary of Expectations* for a complete list of expectations that are currently part of the great expectations vocabulary. Here is a short preview of the glossary:

Table shape

- `expect_column_to_exist`
- `expect_table_columns_to_match_ordered_list`
- `expect_table_row_count_to_be_between`
- `expect_table_row_count_to_equal`

Missing values, unique values, and types

- `expect_column_values_to_be_unique`
- `expect_column_values_to_not_be_null`
- `expect_column_values_to_be_null`
- `expect_column_values_to_be_of_type`
- `expect_column_values_to_be_in_type_list`

Sets and ranges

- `expect_column_values_to_be_in_set`
- `expect_column_values_to_not_be_in_set`
- `expect_column_values_to_be_between`
- `expect_column_values_to_be_increasing`
- `expect_column_values_to_be_decreasing`

String matching

- `expect_column_value_lengths_to_be_between`
- `expect_column_value_lengths_to_equal`
- `expect_column_values_to_match_regex`
- `expect_column_values_to_not_match_regex`
- `expect_column_values_to_match_regex_list`
- `expect_column_values_to_not_match_regex_list`

Datetime and JSON parsing

4. Finalize

Data Docs compiles Expectations and Validations into HTML documentation. By default the HTML website is hosted on your local filesystem. When you are working in a team, the website can be hosted in the cloud (e.g., on S3) and serve as the shared source of truth for the team working on the data pipeline.

To view the expectation suite you just created as HTML, rebuild the data docs and open the website in the browser:

```
# save the Expectation Suite (by default to a JSON file in great_expectations/
↪expectations folder
batch.save_expectation_suite(discard_failed_expectations=False)

# This step is optional, but useful - evaluate the expectations against the current␣
↪batch of data
run_id = datetime.datetime.utcnow().strftime("%Y%m%dT%H%M%S.%fZ")
results = context.run_validation_operator("action_list_operator", assets_to_
↪validate=[batch], run_id=run_id)
expectation_suite_identifier = list(results["details"].keys())[0]
validation_result_identifier = ValidationResultIdentifier(
    expectation_suite_identifier=expectation_suite_identifier,
    batch_identifier=batch.batch_kwargs.to_id(),
    run_id=run_id
)

# Update the Data Docs site to display the new Expectation Suite
# and open the site in the browser
context.build_data_docs()
context.open_data_docs(validation_result_identifier)
```

5.2 Validate Data

Expectations describe Data Assets. Data Assets are composed of Batches. Validation checks Expectations against a Batch of data. Expectation Suites combine multiple Expectations into an overall description of a Batch.

Validation = checking if a Batch of data from a Data Asset X conforms to all Expectations in Expectation Suite Y. Expectation Suite Y is a collection of Expectations that you created that specify what a valid Batch of Data Asset X should look like.

To run Validation you need a **Batch** of data. To get a **Batch** of data you need:

- to provide *batch_kwargs* to a *Data Context*
- to specify an **Expectation Suite** to validate against

This tutorial will explain each of these objects, show how to obtain them, execute validation and view its result.

5.2.1 0. Open Jupyter Notebook

This tutorial assumes that:

- you ran `great_expectations init`
- your current directory is the root of the project where you ran `great_expectations init`

You can either follow the tutorial with the sample National Provider Identifier (NPI) dataset (processed with Pandas) referenced in the *great_expectations init* tutorial, or you can execute the same steps on your project with your own data.

If you get stuck, find a bug or want to ask a question, go to our Slack - this is the best way to get help from the contributors and other users.

Validation is typically invoked inside the code of a data pipeline (e.g., an Airflow operator). This tutorial uses a Jupyter notebook as a validation playground.

The `great_expectations init` command created a `great_expectations/notebooks/` folder in your project. The folder contains example notebooks for pandas, Spark and SQL datasources.

If you are following this tutorial using the NPI dataset, open the pandas notebook. If you are working with your dataset, see the instructions for your datasource:

pandas

```
jupyter notebook great_expectations/notebooks/pandas/validation_playground.ipynb
```

pyspark

```
jupyter notebook great_expectations/notebooks/spark/validation_playground.ipynb
```

SQLAlchemy

```
jupyter notebook great_expectations/notebooks/sql/validation_playground.ipynb
```

5.2.2 1. Get a DataContext Object

A DataContext represents a Great Expectations project. It organizes Datasources, notification settings, data documentation sites, and storage and access for Expectation Suites and Validation Results. The DataContext is configured via a yml file stored in a directory called great_expectations; the configuration file as well as managed Expectation Suites should be stored in version control.

Instantiating a DataContext loads your project configuration and all its resources.

```
context = ge.data_context.DataContext()
```

To read more about DataContexts, see: *DataContexts*

5.2.3 2. Choose an Expectation Suite

The `context` instantiated in the previous section has a convenience method that lists all Expectation Suites created in a project:

```
for expectation_suite_id in context.list_expectation_suites():
    print(expectation_suite_id.expectation_suite_name)
```

Choose the Expectation Suite you will use to validate a Batch of data:

```
expectation_suite_name = "warning"
```

5.2.4 3. Load a batch of data you want to validate

Expectations describe Batches of data - Expectation Suites combine multiple Expectations into an overall description of a Batch. Validation checks a Batch against an Expectation Suite.

For example, a Batch could be the most recent day of log data. For a database table, a Batch could be the data in that table at a particular time.

In order to validate a Batch of data, you will load it as a Great Expectations *Dataset*.

Batches are obtained by using a Data Context's `get_batch` method, which accepts `batch_kwargs` and `expectation_suite_name` as arguments.

Calling this method asks the Context to get a Batch of data using the provided `batch_kwargs` and attach the Expectation Suite `expectation_suite_name` to it.

The `batch_kwargs` argument is a dictionary that specifies a batch of data - it contains all the information necessary for a Data Context to obtain a batch of data from a *Datasource*. The keys of a `batch_kwargs` dictionary will vary depending on the type of Datasource and how it generates Batches, but will always have a `datasource` key with the name of a Datasource. To list the Datasources configured in a project, you may use a Data Context's `list_datasources` method.

pandas

A Pandas Datasource generates Batches from Pandas DataFrames or CSV files. A Pandas Datasource can accept `batch_kwargs` that describe either a path to a file or an existing DataFrame:

```
# list datasources of the type PandasDatasource in your project
[datasource['name'] for datasource in context.list_datasources() if datasource['class_
↪name'] == 'PandasDatasource']
datasource_name = # TODO: set to a datasource name from above

# If you would like to validate a file on a filesystem:
batch_kwargs = {'path': "YOUR_FILE_PATH", 'datasource': datasource_name}

# If you already loaded the data into a Pandas Data Frame:
batch_kwargs = {'dataset': "YOUR_DATAFRAME", 'datasource': datasource_name}

batch = context.get_batch(batch_kwargs, expectation_suite_name)
batch.head()
```

pyspark

A Spark Datasource generates Batches from Spark DataFrames or CSV files. A Spark Datasource can accept `batch_kwargs` that describe either a path to a file or an existing DataFrame:

```
# list datasources of the type SparkDFDatasource in your project
[datasource['name'] for datasource in context.list_datasources() if datasource['class_
↪name'] == 'SparkDFDatasource']
datasource_name = # TODO: set to a datasource name from above

# If you would like to validate a file on a filesystem:
batch_kwargs = {'path': "YOUR_FILE_PATH", 'datasource': datasource_name}
# To customize how Spark reads the file, you can add options under reader_options key␣
↪in batch_kwargs (e.g., header='true')

# If you already loaded the data into a PySpark Data Frame:
batch_kwargs = {'dataset': "YOUR_DATAFRAME", 'datasource': datasource_name}
```

(continues on next page)

(continued from previous page)

```
batch = context.get_batch(batch_kwargs, expectation_suite_name)
batch.head()
```

SQLAlchemy

A SQLAlchemy Datasource generates Batches from tables, views and query results. A SQLAlchemy Datasource can accept `batch_kwargs` that instruct it load a batch from a table, a view, or a result set of a query:

```
# list datasources of the type SqlAlchemyDatasource in your project
[datasource['name'] for datasource in context.list_datasources() if datasource['class_
↪name'] == 'SqlAlchemyDatasource']
datasource_name = # TODO: set to a datasource name from above

# If you would like to validate an entire table or view in your database's default␣
↪schema:
batch_kwargs = {'table': "YOUR_TABLE", 'datasource': datasource_name}

# If you would like to validate an entire table or view from a non-default schema in␣
↪your database:
batch_kwargs = {'table': "YOUR_TABLE", "schema": "YOUR_SCHEMA", 'datasource':␣
↪datasource_name}

# If you would like to validate the result set of a query:
# batch_kwargs = {'query': 'SELECT YOUR_ROWS FROM YOUR_TABLE', 'datasource':␣
↪datasource_name}

batch = context.get_batch(batch_kwargs, expectation_suite_name)
batch.head()
```

The examples of `batch_kwargs` above can also be the outputs of "Generators" used by Great Expectations. You can read about the default Generators' behavior and how to implement additional Generators in this article: *Batch Kwargs Generators*.

5.2.5 4. Validate the batch

When Great Expectations is integrated into a data pipeline, the pipeline calls GE to validate a specific batch (an input to a pipeline's step or its output).

Validation evaluates the Expectations of an Expectation Suite against the given Batch and produces a report that describes observed values and any places where Expectations are not met. To validate the Batch of data call the *`validate()`* method on the batch:

```
validation_result = batch.validate()
```

The `validation_result` object has detailed information about every Expectation in the Expectation Suite that was used to validate the Batch: whether the Batch met the Expectation and even more details if it did not. You can read more about the result object's structure here: *Validation Results*.

You can print this object out:

```
print(json.dumps(validation_result, indent=4))
```

Here is what a part of this object looks like:

```
{
    "results": [
        {
            "success": true,
            "expectation_config": {
                "expectation_type": "expect_column_to_exist",
                "kwargs": {
                    "column": "NPI"
                }
            },
            "exception_info": {
                "raised_exception": false,
                "exception_message": null,
                "exception_traceback": null
            }
        },
        {
            "success": true,
            "expectation_config": {
```


Don't panic! This blob of JSON is meant for machines. *Data Docs* are an compiled HTML view of both expectation suites and validation results that is far more suitable for humans. You will see how easy it is to build them in the next sections.

5.2.6 5. Validation Operators

The `validate()` method evaluates one Batch of data against one Expectation Suite and returns a dictionary of Validation Results. This is sufficient when you explore your data and get to know Great Expectations.

When deploying Great Expectations in a real data pipeline, you will typically discover these additional needs:

- Validating a group of Batches that are logically related (e.g. Did all my Salesforce integrations work last night?).
- Validating a Batch against several Expectation Suites (e.g. Did my nightly clickstream event job have any **critical** failures I need to deal with ASAP or **warnings** I should investigate later?).
- Doing something with the Validation Results (e.g., saving them for a later review, sending notifications in case of failures, etc.).

Validation Operators provide a convenient abstraction for both bundling the validation of multiple Expectation Suites and the actions that should be taken after the validation. See the *Validation Operators And Actions Introduction* for more information.

An instance of `action_list_operator` operator is configured in the default `great_expectations.yml` configuration file. `ActionListValidationOperator` validates each Batch in the list that is passed as `assets_to_validate` argument to its `run` method against the Expectation Suite included within that Batch and then invokes a list of configured actions on every Validation Result.

Below is the operator's configuration snippet in the `great_expectations.yml` file:

```
action_list_operator:
  class_name: ActionListValidationOperator
  action_list:
    - name: store_validation_result
      action:
        class_name: StoreValidationResultAction
    - name: store_evaluation_params
```

(continues on next page)

(continued from previous page)

```
      action:
        class_name: StoreEvaluationParametersAction
    - name: update_data_docs
      action:
        class_name: UpdateDataDocsAction
    - name: send_slack_notification_on_validation_result
      action:
        class_name: SlackNotificationAction
        # put the actual webhook URL in the uncommitted/config_variables.yml file
        slack_webhook: ${validation_notification_slack_webhook}
        notify_on: all # possible values: "all", "failure", "success"
        renderer:
          module_name: great_expectations.render.renderer.slack_renderer
          class_name: SlackRenderer
```

We will show how to use the two most commonly used actions that are available to this operator:

Save Validation Results

The DataContext object provides a configurable `validations_store` where GE can store validation_result objects for subsequent evaluation and review. By default, the DataContext stores results in the `great_expectations/uncommitted/validations` directory. To specify a different directory or use a remote store such as `s3` or `gcs`, edit the stores section of the DataContext configuration object:

```
stores:
  validations_store:
    class_name: ValidationsStore
    store_backend:
      class_name: TupleS3Backend
      bucket: my_bucket
      prefix: my_prefix
```

Removing the store_validation_result action from the `action_list_operator` configuration will disable automatically storing `validation_result` objects.

Send a Slack Notification

The last action in the action list of the Validation Operator above sends notifications using a user-provided callback function based on the validation result.

```
- name: send_slack_notification_on_validation_result
  action:
    class_name: SlackNotificationAction
    # put the actual webhook URL in the uncommitted/config_variables.yml file
    slack_webhook: ${validation_notification_slack_webhook}
    notify_on: all # possible values: "all", "failure", "success"
    renderer:
      module_name: great_expectations.render.renderer.slack_renderer
      class_name: SlackRenderer
```

GE includes a slack-based notification in the base package. To enable a slack notification for results, simply specify the slack webhook URL in the uncommitted/config_variables.yml file:

```
validation_notification_slack_webhook: https://slack.com/your_webhook_url
```

Running the Validation Operator

Before running the Validation Operator, create a `run_id`. A `run_id` links together validations of different data assets, making it possible to track "runs" of a pipeline and follow data assets as they are transformed, joined, annotated, enriched, or evaluated. The run id can be any string; by default, Great Expectations will use an ISO 8601-formatted UTC datetime string.

The default `run_id` generated by Great Expectations is built using the following code:

```
run_id = datetime.datetime.utcnow().strftime("%Y%m%dT%H%M%S.%fZ")
```

When you integrate validation in your pipeline, your pipeline runner probably has a run id that can be inserted here to make smoother integration.

Finally, run the Validation Operator:

```
results = context.run_validation_operator(
    "action_list_operator",
    assets_to_validate=[batch],
    run_id=run_id)
```

5.2.7 6. View the Validation Results in Data Docs

Data Docs compiles raw Great Expectations objects including Expectations and Validations into structured documents such as HTML documentation. By default the HTML website is hosted on your local filesystem. When you are working in a team, the website can be hosted in the cloud (e.g., on S3) and serve as the shared source of truth for the team working on the data pipeline.

Read more about the capabilities and configuration of Data Docs here: *Data Docs*.

One of the actions executed by the validation operator in the previous section rendered the validation result as HTML and added this page to the Data Docs site.

You can open the page programmatically and examine the result:

```
context.open_data_docs()
```

5.2.8 Congratulations!

Now you you know how to validate a Batch of data.

What is next? This is a collection of tutorials that walk you through a variety of useful Great Expectations workflows: *Tutorials*.

5.3 Publishing Data Docs to S3

In this tutorial we will cover publishing a data docs site directly to s3. Publishing a site this way makes reviewing and acting on validation results easy in a team, and provides a central location to review the expectations currently configured for data assets under test.

Configuring data docs requires three simple steps:

1. Configure an S3 bucket.

Modify the bucket name and region for your situation.

```
> aws s3api create-bucket --bucket data-docs.my_org --region us-east-1
{
    "Location": "/data-docs.my_org"
}
```

Configure your bucket policy to enable appropriate access. **IMPORTANT**: your policy should provide access only to appropriate users; data-docs can include critical information about raw data and should generally **not** be publicly accessible. The example policy below **enforces IP-based access**.

Modify the bucket name and IP addresses below for your situation.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "Allow only based on source IP",
    "Effect": "Allow",
    "Principal": "*",
    "Action": "s3:GetObject",
    "Resource": [
      "arn:aws:s3:::data-docs.my_org",
      "arn:aws:s3:::data-docs.my_org/*"
    ],
    "Condition": {
      "IpAddress": {
        "aws:SourceIp": [
          "192.168.0.1/32",
          "2001:db8:1234:1234::/64"
        ]
      }
    }
  }
  ]
}
```

Modify the policy above and save it to a file called *ip-policy.json* in your local directory. Then, run:

```
> aws s3api put-bucket-policy --bucket data-docs.my_org --policy file://ip-policy.json
```

2. Edit your *great_expectations.yml* file to change the *data_docs_sites* configuration for the site you will publish. **Add the `s3_site`** section below existing site configuration.

```
# ... additional configuration above
data_docs_sites:
  local_site:
    class_name: SiteBuilder
    store_backend:
```

(continues on next page)

(continued from previous page)

```
      class_name: TupleFilesystemStoreBackend
      base_directory: uncommitted/data_docs/local_site/
  s3_site:
    class_name: SiteBuilder
    store_backend:
      class_name: TupleS3StoreBackend
      bucket: data-docs.my_org  # UPDATE the bucket name here to match the bucket you␣
↪configured above.
# ... additional configuration below
```

3. Build your documentation:

```
> great_expectations docs build
Building...
```

You're now ready to visit the site! Your site will be available at the following URL: http://data-docs.my_org.s3.amazonaws.com/index.html

5.3.1 Additional Resources

Optionally, you may wish to update static hosting settings for your bucket to enable AWS to automatically serve your index.html file or a custom error file:

```
> aws s3 website s3://data-docs.my_org/ --index-document index.html
```

For more information on static site hosting in AWS, see the following:

- AWS Website Hosting
- AWS Static Site Access Permissions
- AWS Website configuration

5.4 Saving Metrics

Saving metrics during Validation makes it easy to construct a new data series based on observed dataset characteristics computed by Great Expectations. That data series can serve as the source for a dashboard or overall data quality metrics, for example.

Storing metrics is still a **beta** feature of Great Expectations, and we expect configuration and capability to evolve rapidly.

5.4.1 Adding a MetricsStore

A MetricStore is a special store that can store Metrics computed during Validation. A Metric store tracks the run_id of the validation and the expectation suite name in addition to the metric name and metric kwargs.

In most cases, a MetricStore will be configured as a SQL database. To add a MetricStore to your DataContext, add the following yaml block to the "stores" section:

```
stores:
    #  ...
    metrics_store:  # You can choose any name for your metric store
```

(continues on next page)

(continued from previous page)

```
        class_name: MetricStore
        store_backend:
            class_name: DatabaseStoreBackend
            # These credentials can be the same as those used in a Datasource␣
↪configuration
            credentials: ${my_store_credentials}
```

The next time your DataContext is loaded, it will connect to the database and initialize a table to store metrics if one has not already been created. See the *Metrics Reference* for more information on additional configuration options.

5.4.2 Configuring a Validation Action

Once a MetricStore is available, it is possible to configure a new *StoreMetricsAction* to save metrics during validation.

Add the following yaml block to your DataContext validation operators configuration:

```
validation_operators:
  # ...
  action_list_operator:
    class_name: ActionListValidationOperator
    action_list:
      # ...
      - name: store_metrics
        action:
          class_name: StoreMetricsAction
          target_store_name: metrics_store  # This should match the name of the store␣
↪configured above
          # Note that the syntax for selecting requested metrics will change in a␣
↪future release
          requested_metrics:
            *:  # The asterisk here matches *any* expectation suite name
              # use the 'kwargs' key to request metrics that are defined by kwargs,
              # for example because they are defined only for a particular column
              # - column:
              #     Age:
              #       - expect_column_min_to_be_between.result.observed_value
              - statistics.evaluated_expectations
              - statistics.successful_expectations
```

The *StoreMetricsValidationAction* processes an *ExpectationValidationResult* and stores Metrics to a configured Store. Now, when your operator is executed, the requested metrics will be available in your database!

```
context.run_validation_operator('action_list_operator', (batch_kwargs, expectation_
↪suite_name))
```

CHAPTER

SIX

FEATURES

These pages provide a high-level overview of key great expectations features, with an emphasis on explaining the thinking behind the tools. See the *Reference* section for more detailed configuration and usage guides, and our blog for more information on how we think about data.

6.1 Expectations

Expectations are the workhorse abstraction in Great Expectations. Like assertions in traditional python unit tests, Expectations provide a flexible, declarative language for describing expected behavior. Unlike traditional unit tests, Great Expectations applies Expectations to data instead of code.

Expectations *enhance communication* about your data and *amplify quality* in data applications. Using expectations helps reduce trips to domain experts and avoids leaving insights about data on the "cutting room floor."

6.1.1 How to build expectations

Generating expectations is one of the most important parts of using Great Expectations effectively, and there are a variety of methods for generating and encoding expectations. When expectations are encoded in the GE format, they become shareable and persistent sources of truth about how data was expected to behave-and how it actually did.

There are several paths to generating expectations:

1. Automated inspection of datasets. Currently, the profiler mechanism in GE produces expectation suites that can be used for validation. In some cases, the goal is *Profiling* your data, and in other cases automated inspection can produce expectations that will be used in validating future batches of data.
2. Expertise. Rich experience from Subject Matter Experts, Analysts, and data owners is often a critical source of expectations. Interviewing experts and encoding their tacit knowledge of common distributions, values, or failure conditions can be can excellent way to generate expectations.
3. Exploratory Analysis. Using GE in an exploratory analysis workflow (e.g. within Jupyter notebooks) is an important way to develop experience with both raw and derived datasets and generate useful and testable expectations about characteristics that may be important for the data's eventual purpose, whether reporting or feeding another downstream model or data system.

Expectations come to your data

Great Expectations's connect-and-expect API makes it easy to declare Expectations within the tools you already use for data exploration: jupyter notebooks, the ipython console, scratch scripts, etc.

```
>>> import great_expectations as ge
>>> my_df = ge.read_csv("./tests/examples/titanic.csv")
>>> my_df.expect_column_values_to_be_in_set("Sex", ["male", "female"])
{
    'success': True,
    'summary_obj': {
        'unexpected_count': 0,
        'unexpected_percent': 0.0,
        'unexpected_percent_nonmissing': 0.0,
        'partial_unexpected_list': []
    }
}
```

Instant feedback

When you invoke an Expectation method from a notebook or console, it will immediately return a dictionary containing the result and a list of exceptions.

For example:

```
>> print my_df.PClass.value_counts()
3rd    711
1st    322
2nd    279
*        1
Name: PClass, dtype: int64

>> my_df.expect_column_values_to_be_in_set(
    "PClass",
    ["1st", "2nd", "3rd"]
)
{
    'success': False,
    'summary_obj': {
        'unexpected_count': 1,
        'unexpected_percent': 0.0007616146230007616,
        'unexpected_percent_nonmissing': 0.0007616146230007616,
        'partial_unexpected_list': ['*']
    }
}
```

Another example:

```
 >> my_df.expect_column_values_to_match_regex(
     "Name",
     "^[A-Za-z\, \(\)\']+$"
 )
 {
     'success': False,
     'summary_obj': {
         'unexpected_count': 16,
```

(continues on next page)

(continued from previous page)

```
        'unexpected_percent': 0.012185833968012186,
        'unexpected_percent_nonmissing': 0.012185833968012186,
        'partial_unexpected_list': [
            'Bjornstrm-Steffansson, Mr Mauritz Hakan',
            'Brown, Mrs James Joseph (Margaret Molly" Tobin)"',
            'Frolicher-Stehli, Mr Maxmillian',
            'Frolicher-Stehli, Mrs Maxmillian (Margaretha Emerentia Stehli)',
            'Lindeberg-Lind, Mr Erik Gustaf',
            'Roebling, Mr Washington Augustus 2nd',
            'Rothes, the Countess of (Noel Lucy Martha Dyer-Edwardes)',
            'Simonius-Blumer, Col Alfons',
            'Thorne, Mr George (alias of: Mr George Rosenshine)',
            'Downton (?Douton), Mr William James',
            'Aijo-Nirva, Mr Isak',
            'Johannesen-Bratthammer, Mr Bernt',
            'Larsson-Rondberg, Mr Edvard',
            'Nicola-Yarred, Miss Jamila',
            'Nicola-Yarred, Master Elias',
            'Thomas, Mr John (? 1st/2nd class)'
        ]
    }
}
```

This instant feedback helps you zero in on exceptions very quickly, taking a lot of the pain and guesswork out of early data exploration.

Iterative exploratory analysis

Build expectations as you conduct exploratory data analysis to ensure insights about data processes and pipelines remain part of your team's knowldege. Great Expectations's library of Expectations has been developed by a broad cross-section of data scientists and engineers. Check out the *Glossary of Expectations*; it covers all kinds of practical use cases:

- Foreign key verification and row-based accounting for ETL
- Form validation and regex pattern-matching for names, URLs, dates, addresses, etc.
- Checks for missing data
- Crosstabs
- Distributions for statistical modeling.
- etc.

You can also add notes or even structured metadata to expectations to describe the intent of an expectation or anything else relevant for understanding it:

```
>> my_df.expect_column_values_to_match_regex(
    "Name",
    "^[A-Za-z\, \(\)\']+$",
    meta = {
        "notes": "A simple experimental regex for name matching.",
        "source": "max@company.com"
        }
   )
```

6.2 Validation Operators And Actions Introduction

The *validate* method evaluates one batch of data against one expectation suite and returns a dictionary of validation results. This is sufficient when you explore your data and get to know Great Expectations. When deploying Great Expectations in a real data pipeline, you will typically discover additional needs:

- validating a group of batches that are logically related
- validating a batch against several expectation suites
- doing something with the validation results (e.g., saving them for a later review, sending notifications in case of failures, etc.).

Validation Operators provide a convenient abstraction for both bundling the validation of multiple expectation suites and the actions that should be taken after the validation.

6.2.1 Finding a Validation Operator that is right for you

Each Validation Operator encodes a particular set of business rules around validation. Currently, two Validation Operator implementations are included in the Great Expectations distribution.

The classes that implement them are in `great_expectations.validation_operators` module.

ActionListValidationOperator

ActionListValidationOperator validates each batch in the list that is passed as *assets_to_validate* argument to its *run* method against the expectation suite included within that batch and then invokes a list of configured actions on every validation result.

Actions are Python classes with a *run* method that takes a result of validating a batch against an expectation suite and does something with it (e.g., save validation results to the disk or send a Slack notification). Classes that implement this API can be configured to be added to the list of actions for this operator (and other operators that use actions). Read more about actions here: *Actions*.

An instance of this operator is included in the default configuration file *great_expectations.yml* that *great_expectations init* command creates.

A user can choose the actions to perform in their instance of the operator.

Read more about ActionListValidationOperator here: *ActionListValidationOperator*.

WarningAndFailureExpectationSuitesValidationOperator

WarningAndFailureExpectationSuitesValidationOperator implements a business logic pattern that many data practitioners consider useful.

It validates each batch of data against two different expectation suites: “failure” and “warning”. Group the expectations that you create about a data asset into two expectation suites. The critical expectations that you are confident that the data should meet go into the expectation suite that this operator calls “failure” (meaning, not meeting these expectations should be considered a failure in the pipeline). The rest of expectations go into the “warning” expectation suite.

The failure Expectation Suite contains expectations that are considered important enough to justify stopping the pipeline when they are violated.

WarningAndFailureExpectationSuitesValidationOperator retrieves two expectation suites for every data asset in the *assets_to_validate* argument of its *run* method.

After completing all the validations, it sends a Slack notification with the success status. Note that it doesn't use an Action to send its Slack notification, because the notification has also been customized to summarize information from both suites.

Read more about WarningAndFailureExpectationSuitesValidationOperator here: *WarningAndFailureExpectationSuitesValidationOperator*

6.2.2 Implementing Custom Validation Operators

If you wish to implement some validation handling logic that is not supported by the operators included in Great Expectations, follow these steps:

- Extend the `great_expectations.validation_operators.ValidationOperator` base class
- Implement the *run* method in your new class
- Put your class in the plugins directory of your project (see *plugins_directory* property in the *great_expectations.yml configuration file*)

Once these steps are complete, your new Validation Operator can be configured and used.

If you think that the business logic of your operator can be useful to other data practitioners, please consider contributing it to Great Expectations.

6.2.3 Configuring and Running a Validation Operator

If you are using a Validation Operator that came with GE or was contributed by another developer, you can get to a rich set of useful behaviors with very little coding. This is done by editing the operator's configuration in the GE configuration file and by extending the operator in case you want to add new behavior.

To use a Validation Operator (one that is included in Great Expectations or one that you implemented in your project's plugins directory), you need to configure an instance of the operator in your *great_expectations.yml* file and then invoke this instance from your Python code.

Configuring an operator

All Validation Operators configuration blocks should appear under *validation_operators* key in *great_expectations.yml*

To configure an instance of an operator in your Great Expectations context, create a key under *validation_operators*. This is the name of you will use to invoke this operator.

```
my_operator:
    class_name: TheClassThatImplementsMyOperator
    module_name: thefilethatyoutheclassisin
    foo: bar
```

In the example of an operator config block above:

- the *class_name* value is the name of the class that implements this operator.
- the key *module_name* must be specified if the class is not in the default module.
- the *foo* key specifies the value of the *foo* argument of this class' constructor. Since every operator class might define its own constructor, the keys will vary.

Invoking an operator

This is an example of invoking an instance of a Validation Operator from Python:

```
results = context.run_validation_operator(
    assets_to_validate=[batch0, batch1, ...],
    run_id="some_string_that_uniquely_identifies_this_run",
    validation_operator_name="perform_action_list_operator",
)
```

- *assets_to_validate* - an iterable that specifies the data assets that the operator will validate. The members of the list can be either batches or triples that will allow the operator to fetch the batch: (data_asset_name, expectation_suite_name, batch_kwargs) using this method: *get_batch()*
- run_id - pipeline run id, a timestamp or any other string that is meaningful to you and will help you refer to the result of this operation later
- validation_operator_name you can instances of a class that implements a Validation Operator

Each operator class is free to define its own object that the *run* method returns. Consult the reference of the specific Validation Operator.

6.2.4 Actions

The Validation Operator implementations above invoke actions.

An action is a way to take an arbitrary method and make it configurable and runnable within a data context.

The only requirement from an action is for it to have a take_action method.

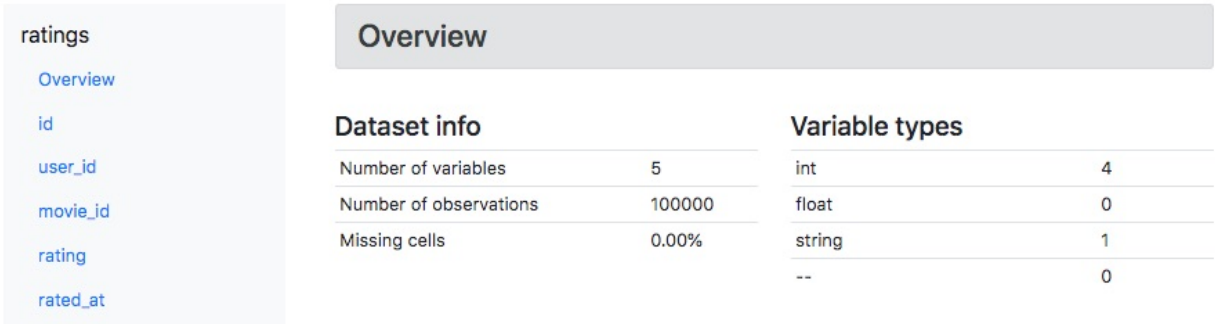
GE comes with a list of actions that we consider useful and you can reuse in your pipelines. Most of them take in validation results and do something with them.

6.3 Data Docs

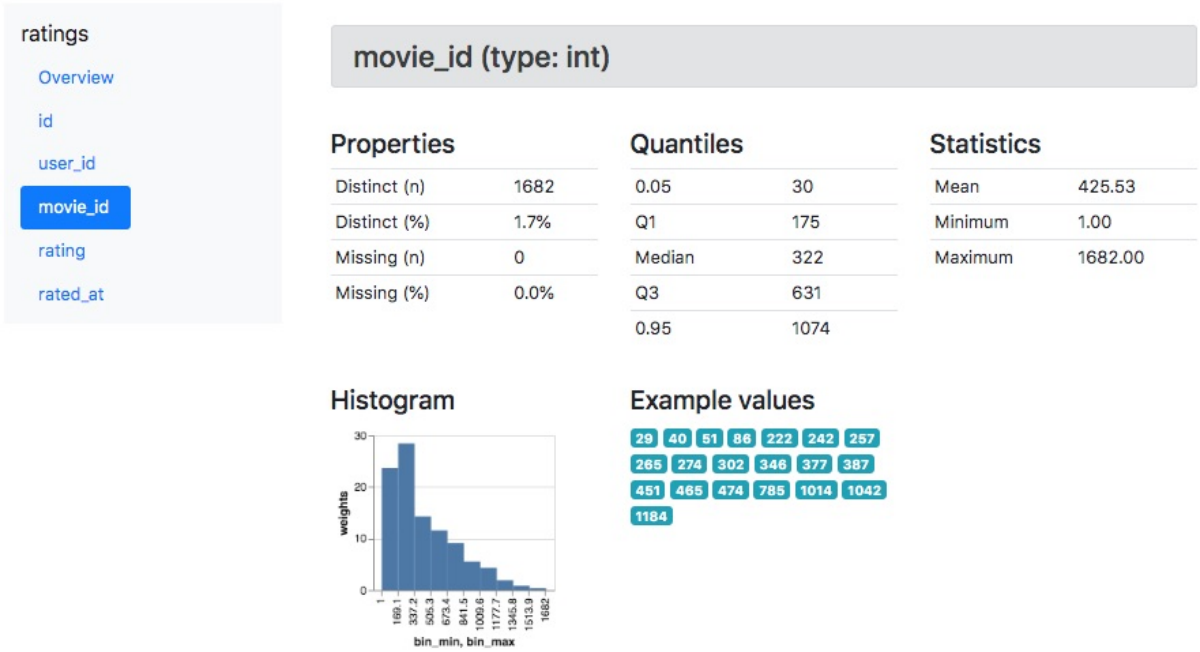
Data Docs compiles raw Great Expectations objects including Expectations and Validations into structured documents such as HTML documentation that display key characteristics of a dataset.

HTML documentation takes expectation suites and validation results and produces clear, functional, and self-healing documentation of expected and observed data characteristics. Together with profiling, it can help to rapidly create a clearer picture of your data, and keep your entire team on the same page as data evolves.

For example, the default BasicDatasetProfiler in GE will produce validation_results which compile to a page for each table or DataFrame including an overview section:



And then detailed statistics for each column:



The Great Expectations DataContext uses a configurable "data documentation site" to define which artifacts to compile and how to render them as documentation. Multiple sites can be configured inside a project, each suitable for a particular data documentation use case.

For example, we have identified three common use cases for using documentation in a data project. They are to:

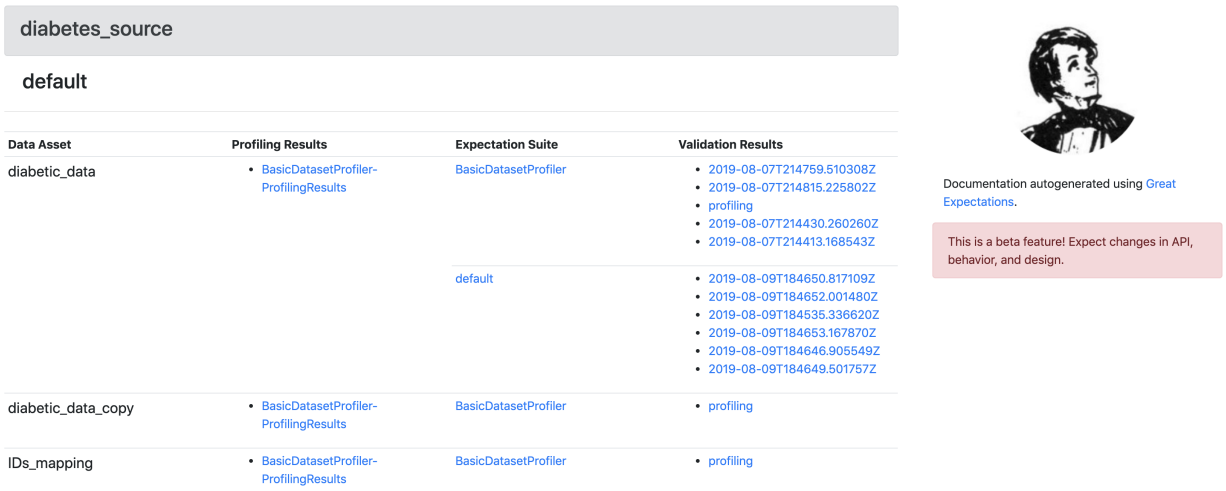
1. Visualize all Great Expectations artifacts from the local repository of a project as HTML: expectation suites, validation results and profiling results.

2. Maintain a "shared source of truth" for a team working on a data project. Such documentation renders all the artifacts committed in the source control system (expectation suites and profiling results) and a continuously updating data quality report, built from a chronological list of validations by run id.

3. Share a spec of a dataset with a client or a partner. This is similar to API documentation in software development. This documentation would include profiling results of the dataset to give the reader a quick way to grasp what the data looks like, and one or more expectation suites that encode what is expected from the data to be considered valid.

To support these (and possibly other) use cases Great Expectations has a concept of "data documentation site". Multiple sites can be configured inside a project, each suitable for a particular data documentation use case.

Here is an example of a site:



The behavior of a site is controlled by configuration in the DataContext's great_expectations.yml file.

Users can specify

- which datasources to document (by default, all)
- whether to include expectations, validations and profiling results sections
- where the expectations and validations should be read from (filesystem, S3, or GCS)
- where the HTML files should be written (filesystem, S3, or GCS)
- which renderer and view class should be used to render each section

6.3.1 Customizing HTML Documentation

The HTML documentation generated by Great Expectations Data Docs is fully customizable. If you would like to customize the look and feel of these pages or create your own, see *Customizing Data Docs*.

See the *Data Docs Reference* for more information.

6.4 DataContexts

A DataContext represents a Great Expectations project. It organizes storage and access for expectation suites, datasources, notification settings, and data fixtures.

The DataContext is configured via a yml file stored in a directory called great_expectations; the configuration file as well as managed expectation suites should be stored in version control.

DataContexts manage connections to your data and compute resources, and support integration with execution frameworks (such as airflow, Nifi, dbt, or dagster) to describe and produce batches of data ready for analysis. Those features enable fetching, validation, profiling, and documentation of your data in a way that is meaningful within your existing infrastructure and work environment.

DataContexts also manage Expectation Suites. Expectation Suites combine multiple Expectation Configurations into an overall description of a dataset. Expectation Suites should have names corresponding to the kind of data they define, like "NPI" for National Provider Identifier data or "company.users" for a users table.

The DataContext also provides other services, such as storing and substituting evaluation parameters during validation. See *Data Context Evaluation Parameter Store* for more information.

See the *Data Context Reference* for more information.

6.5 Datasources

Datasources are responsible for connecting data and compute infrastructure. Each Datasource provides Great Expectations Data Assets connected to a specific compute environment, such as a SQL database, a Spark cluster, or a local in-memory Pandas DataFrame. Datasources know how to access data from relevant sources such as an existing object from a DAG runner, a SQL database, an S3 bucket, GCS, or a local filesystem.

To bridge the gap between those worlds, Datasources can interact closely with *Batch Kwargs Generators* which are aware of a source of data and can produce produce identifying information, called "batch_kwargs" that datasources can use to get individual batches of data.

See *Datasource Reference* for more detail about configuring and using datasources in your DataContext.

See datasource module docs *Datasource Module* for more detail about available datasources.

6.6 Validation

Once you've constructed and stored Expectations, you can use them to validate new data. Validation generates a report that details any specific deviations from expected values.

We recommend using *DataContexts* to manage expectation suites and coordinate validation across runs.

6.6.1 Validation Results

The report contains information about:

- the overall sucess (the *success* field),
- summary statistics of the expectations (the *statistics* field), and
- the detailed results of each expectation (the *results* field).

An example report looks like the following:

```
>> import json
>> import great_expectations as ge
>> my_expectation_suite = json.load(file("my_titanic_expectations.json"))
>> my_df = ge.read_csv(
    "./tests/examples/titanic.csv",
    expectation_suite=my_expectation_suite
)
>> my_df.validate()

{
  "results" : [
    {
      "expectation_type": "expect_column_to_exist",
      "success": True,
      "kwargs": {
        "column": "Unnamed: 0"
      }
    },
    ...
```

(continues on next page)

(continued from previous page)


```
    {
      "unexpected_list": 30.397989417989415,
      "expectation_type": "expect_column_mean_to_be_between",
      "success": True,
      "kwargs": {
        "column": "Age",
        "max_value": 40,
        "min_value": 20
      }
    },
    {
      "unexpected_list": [],
      "expectation_type": "expect_column_values_to_be_between",
      "success": True,
      "kwargs": {
        "column": "Age",
        "max_value": 80,
        "min_value": 0
      }
    },
    {
      "unexpected_list": [
        "Downton (?Douton), Mr William James",
        "Jacobsohn Mr Samuel",
        "Seman Master Betros"
      ],
      "expectation_type": "expect_column_values_to_match_regex",
      "success": True,
      "kwargs": {
        "regex": "[A-Z][a-z]+(?: \\([A-Z][a-z]+\\))?, ",
        "column": "Name",
        "mostly": 0.95
      }
    },
    {
      "unexpected_list": [
        "*"
      ],
      "expectation_type": "expect_column_values_to_be_in_set",
      "success": False,
      "kwargs": {
        "column": "PClass",
        "value_set": [
          "1st",
          "2nd",
          "3rd"
        ]
      }
    }
  ],
  "success", False,
  "statistics": {
      "evaluated_expectations": 10,
      "successful_expectations": 9,
      "unsuccessful_expectations": 1,
      "success_percent": 90.0,
  }
```


(continues on next page)

(continued from previous page)

```
}
```

6.6.2 Reviewing Validation Results

The easiest way to review Validation Results is to view them from your local Data Docs site, where you can also conveniently view Expectation Suites and with additional configuration, Profiling Results (see *Data Docs Site Configuration*). Out of the box, Great Expectations Data Docs is configured to compile a local data documentation site when you start a new project by running `great_expectations init`. By default, this local site is saved to the `uncommitted/data_docs/local_site/` directory of your project and will contain pages for Expectation Suites and Validation Results.

If you would like to review the raw validation results in JSON format, the default Validation Results directory is `uncommitted/validations/`. Note that by default, Data Docs will only compile Validation Results located in this directory.

To learn more about setting up Great Expectations for your team read *Using Great Expectations on Teams*.

6.6.3 Validation Operators

The example above demonstrates how to validate one batch of data against one expectation suite. The *validate* method returns a dictionary of validation results. This is sufficient when exploring your data and getting to know Great Expectations. When deploying Great Expectations in a real data pipeline, you will typically discover additional needs:

- validating a group of batches that are logically related
- validating a batch against several expectation suites
- doing something with the validation results (e.g., saving them for a later review, sending notifications in case of failures, etc.).

Validation Operators are mini-applications that can be configured to implement these scenarios.

Read *Validation Operators And Actions Introduction* to learn more.

6.6.4 Deployment patterns

Useful deployment patterns include:

- Include validation at the end of a complex data transformation, to verify that no cases were lost, duplicated, or improperly merged.
- Include validation at the *beginning* of a script applying a machine learning model to a new batch of data, to verify that its distributed similarly to the training and testing set.
- Automatically trigger table-level validation when new data is dropped to an FTP site or S3 bucket, and send the validation report to the uploader and bucket owner by email.
- Schedule database validation jobs using cron, then capture errors and warnings (if any) and post them to Slack.
- Validate as part of an Airflow task: if Expectations are violated, raise an error and stop DAG propagation until the problem is resolved. Alternatively, you can implement expectations that raise warnings without halting the DAG.

For certain deployment patterns, it may be useful to parameterize expectations, and supply evaluation parameters at validation time. See *Evaluation Parameters* for more information.

6.7 Metrics

Metrics are values derived from one or more Data Assets that can be used to evaluate expectations or to summarize the result of Validation. A Metric is obtained from an ExpectationValidationResult or ExpectationSuiteValidationResult by providing the *metric_name* and *metric_kwargs* or *metric_kwargs_id*.

A metric name is a dot-delimited string that identifies the value, such as *expect_column_values_to_be_unique .success* or *expect_column_values_to_be_between.result.unexpected_percent*.

Metric Kwargs are key-value pairs that identify the metric within the context of the validation, such as "column": "Age". Different metrics may require different Kwargs.

A metric_kwargs_id is a string representation of the Metric Kwargs that can be used as a database key. For simple cases, it could be easily readable, such as *column=Age*, but when there are multiple keys and values or complex values, it will most likely be an md5 hash of key/value pairs. It can also be None in the case that there are no kwargs required to identify the metric.

See the *Metrics Reference* or *Saving Metrics Tutorial* for more information.

Last updated: **|lastupdate|**

6.8 Custom expectations

Expectations are especially useful when they capture critical aspects of data understanding that analysts and practitioners know based on its *semantic* meaning. It's common to want to extend Great Expectations with application- or domain-specific Expectations. For example:

```
expect_column_text_to_be_in_english
expect_column_value_to_be_valid_icd_code
```

These Expectations aren't included in the default set, but could be very useful for specific applications.

Fear not! Great Expectations is designed for customization and extensibility.

Building custom expectations is easy and allows your custom logic to become part of the validation, documentation, and even profiling workflows that make Great Expectations stand out. See the guide on *Building Custom Expectations* for more information on building expectations and updating DataContext configurations to automatically load batches of data with custom Data Assets.

6.9 Batch Kwargs Generators

Batch Kwargs are specific instructions for a Datasource about what data should be prepared as a "batch" for validation. The batch could be a specific database table, the most recent log file delivered to S3, or even a subset of one of those objects such as the first 10,000 rows.

A BatchKwargsGenerator builds those instructions for GE datasources by inspecting storage backends or data, or by maintaining configuration such as commonly-used paths or filepath conventions. That allows BatchKwargsGenerators to add flexibility in how to obtain data such as by exposing time-based partitions or sampling data.

For example, a Batch Kwargs Generator could be **configured** to produce a SQL query that logically represents "rows in the Events table with a type code of 'X' that occurred within seven days of a given timestamp." With that configuration, you could provide a timestamp as a partition name, and the Batch Kwargs Generator will produce instructions that a SQLAlchemyDatasource could use to materialize a SQLAlchemyDataset corresponding to that batch of data and ready for validation.

6.9.1 Batch

A batch is a sample from a data asset, sliced according to a particular rule. For example, an hourly slide of the Events table or "most recent *users* records."

A Batch is the primary unit of validation in the Great Expectations DataContext. Batches include metadata that identifies how they were constructed–the same "batch_kwargs" assembled by the generator, "batch_markers" that provide more detailed metadata to aid in replicating complicated workflows, and optionally "batch_parameters" that include information such as an asset or partition name.

See more detailed documentation on the *Generator Module*.

6.10 Using Great Expectations on Teams

Now that you've enjoyed single player mode, let's bring Great Expectations to your team.

When you move from single user to collaborative use there are a few major things to consider.

6.10.1 Major Considerations

Where Should Expectations Live?

If you followed our best practice recommendation of committing the `great_expectations` directory to your source control repository, then this question is already answered! Expectations live right in your repo!

It is also possible to store expectations on cloud storage providers–we recommend enabling versioning in that case. See the documentation on **:ref:`customizing the data docs store backend <customizing_data_docs_store_backend>`_** for more information, or follow the *tutorial*.

Where Should Validations Live?

When using the default Validation Operators, Validations are stored in your `great_expectations/uncommitted/validations/` directory. Because these may include examples of data (which could be sensitive or regulated) these Validations **should not** be committed to a source control system.

You can configure a Store to write these to a cloud provider blob storage such as Amazon S3, Google Cloud Storage, or some other securely mounted file system. This will depend on your team's deployment patterns.

Where Should Data Docs Live?

Similar to Validations, Data Docs are by default stored in your `great_expectations/uncommitted/data_docs/` directory. Because these may include examples of data (which could be sensitive or regulated) these **should not** be committed to a source control system.

You can configure a store to write these to a cloud provider blob storage such as Amazon S3, Google Cloud Storage, or some other securely mounted file system.

See the *data docs reference* for more information on configuring data docs to use cloud storage, or follow the **:ref:`tutorial <publishing_data_docs_to_s3>`_** to configure a site now.

Where Should Notifications Go?

Some teams enjoy realtime data quality alerts in Slack. We find setting up a channel with an obvious name like `data-qualty-notifications` a nice place to have Great Expectations post to.

6.10.2 How Do You On-board a New Teammate?

If you have a project in a repo that includes your `great_expectations` directory, onboarding is simple! Your teammates will need to:

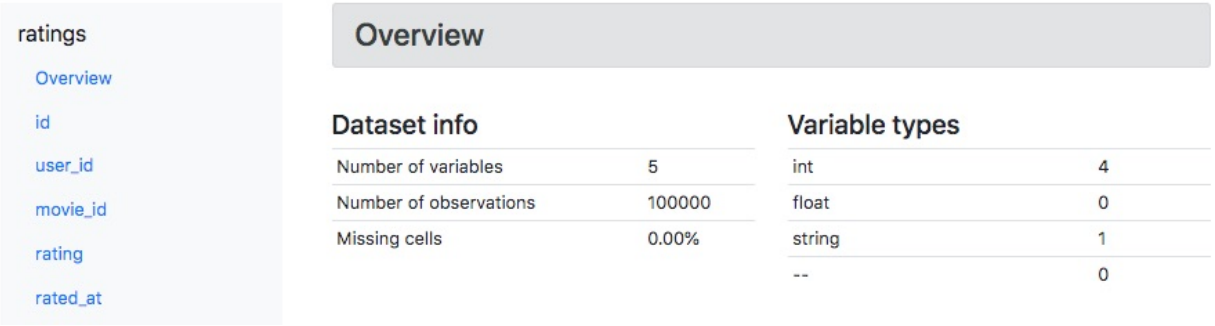
1. Clone the repo.
2. Run `great_expectations init` to create any missing directories.
3. Add any secrets required by your datasources to the file `great_expectations/uncommitted/config_variables.yml`.

6.11 Profiling

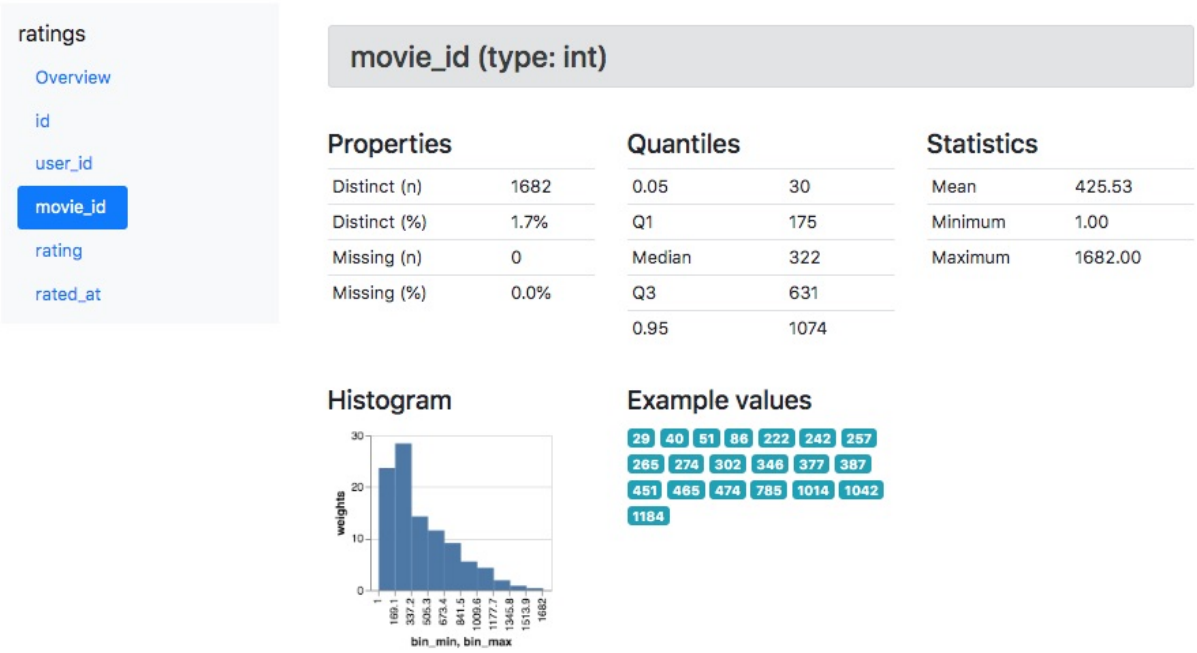
Profiling is a way of Rendering Validation Results to produce a summary of observed characteristics. When Validation Results are rendered as Profiling data, they create a new section in *Data Docs*. By computing the **observed** properties of data, Profiling helps to understand and reason about the data's **expected** properties.

To produce a useful data overview, Great Expectations uses a *profiler* to build a special Expectation Suite. Unlike the Expectations that are typically used for data validation, expectations for Profiling do not necessarily apply any constraints. They can simply identify statistics or other data characteristics that should be evaluated and made available in Great Expectations. For example, when the included `BasicDatasetProfiler` encounters a numeric column, it will add an `expect_column_mean_to_be_between` expectation but choose the min_value and max_value to both be None: essentially only saying that it expects a mean to exist.

The default BasicDatasetProfiler will thus produce a page for each table or DataFrame including an overview section:



And then detailed statistics for each column:



Profiling is still a beta feature in Great Expectations. Over time, we plan to extend and improve the `BasicDatasetProfiler` and also add additional profilers.

Warning: `BasicDatasetProfiler` will evaluate the entire batch without limits or sampling, which may be very time consuming. As a rule of thumb, we recommend starting with small batches of data.

See the *Profiling Reference* for more information.

6.12 Profilers

Great Expectations provides a mechanism to automatically generate expectations, using a feature called a *Profiler*. A Profiler builds an Expectation Suite from one or more Data Assets. It usually also validates the data against the newly-generated Expectation Suite to return a Validation Result. There are several Profilers included with Great Expectations.

A Profiler makes it possible to quickly create a starting point for generating expectations about a Dataset. For example, during the *init* flow, Great Expectations uses the *SampleExpectationsDatasetProfiler* to demonstrate important features of Expectations by creating and validating an Expectation Suite that has several different kinds of expectations built from a small sample of data. A Profiler is also critical to generating the Expectation Suites used during *Profiling*.

You can also extend Profilers to capture organizational knowledge about your data. For example, a team might have a convention that all columns **named** "id" are primary keys, whereas all columns ending with the **suffix** "_id" are foreign keys. In that case, when the team using Great Expectations first encounters a new dataset that followed the convention, a Profiler could use that knowledge to add an expect_column_values_to_be_unique Expectation to the "id" column (but not, for example an "address_id" column).

Last updated: **|lastupdate|**

CHAPTER

SEVEN

REFERENCE

Reference materials provide clear descriptions of GE features. See the *Module docs* for more detailed descriptions of classes and methods.

7.1 Basic Functionality

7.1.1 Creating Expectations

Creating expectations involves generating and saving an expectation suite. This reference assumes familiarity with our general philosophy for using expectations outlined in the *Expectations* feature guide.

Expectation Creation Notebooks

Saving expectations

At the end of your exploration, call *save_expectation_suite* to store all Expectations from your session to your pipeline test files. *By default, when you save an expectation suite, only expectations that succeeded on their last run are included in the saved suite.*

This is how you always know what to expect from your data.

```
>> my_df.save_expectation_suite("my_titanic_expectations.json")
```

For more detail on how to control expectation output, please see *Standard arguments for expectations* and *result_format*.

7.1.2 Distributional Expectations

Distributional expectations help identify when new datasets or samples may be different than expected, and can help ensure that assumptions developed during exploratory analysis still hold as new data becomes available. You should use distributional expectations in the same way as other expectations: to help accelerate identification of risks as diverse as changes in a system being modeled or disruptions to a complex upstream data feed.

Great Expectations' Philosophy of Distributional Expectations

Great Expectations attempts to provide a simple, expressive framework for describing distributional expectations. The framework generally adopts a nonparametric approach, although it is possible to build expectations from parameterized distributions.

The design is motivated by the following assumptions:

- Encoding expectations into a simple object that allows for portable data pipeline testing is the top priority. In many circumstances the loss of precision associated with "compressing" data into an expectation may be beneficial because of its intentional simplicity as well as because it adds a very light layer of obfuscation over the data which may align with privacy preservation goals.
- While it should be possible to easily extend the framework with more rigorous statistical tests, great expectations should provide simple, reasonable defaults. Care should be taken in cases where robust statistical guarantees are expected.
- Building and interpreting expectations should be intuitive: a more precise partition object implies a more precise expectation.

Partition Objects

The core constructs of a great expectations distributional expectation are the partition and associated weights.

For continuous data:

- A partition is defined by an ordered list of points that define intervals on the real number line. Note that partition intervals do not need to be uniform.
- Each bin in a partition is partially open: a data element x is in bin i if lower_bound_i <= x < upper_bound_i.
- However, following the behavior of numpy.histogram, a data element x is in the largest bin k if x == upper_bound_k.
- A bin may include -Infinity and Infinity as endpoints, however, those endpoints are not supported by the Kolmogorov-Smirnov test.
- Partition weights define the probability of the associated interval. Note that this effectively applies a "piecewise uniform" distribution to the data for the purpose of statistical tests. The weights must define a valid probability distribution, ie they must be non-negative numbers that sum to 1.

Example continuous partition object:

```
>>> json.dumps(partition, indent=2)
{
  "bins": [
    0,
    1,
    2,
    10
  ],
  "weights": [
    0.3,
    0.3,
    0.4
  ]
}
```

For discrete/categorical data:

- A partition defines the categorical values present in the data.

- Partition weights define the probability of the associated categorical value.

Example discrete partition object:

```
{
  "values": [ "cat", "dog", "fish"],
  "weights": [0.3, 0.3, 0.4]
}
```

Constructing Partition Objects

Three convenience functions are available to easily construct partition objects from existing data:

- *continuous_partition_data*
- *categorical_partition_data*
- *kde_partition_data*

Convenience functions are also provided to validate that an object is a valid partition density object:

- *is_valid_continuous_partition_object*
- *is_valid_categorical_partition_object*

Tests interpret partition objects literally, so care should be taken when a partition includes a segment with zero weight. The convenience methods consequently allow you to include small amounts of residual weight on the "tails" of a dataset used to construct a partition.

Distributional Expectations Core Tests

Distributional expectations rely on three tests for their work.

Kullback-Leibler (KL) divergence is available as an expectation for both categorical and continuous data (continuous data will be discretized according to the provided partition prior to computing divergence). Unlike KS and Chi-Squared tests which can use a p-value, you must provide a threshold for the relative entropy to use KL divergence. Further, KL divergence is not symmetric.

- *expect_column_kl_divergence_to_be_less_than*

For continuous data, the expect_column_bootstrapped_ks_test_p_value_to_be_greater_than expectation uses the Kolmogorov-Smirnov (KS) test, which compares the actual and expected cumulative densities of the data. Because of the partition_object's piecewise uniform approximation of the expected distribution, the test would be overly sensitive to differences when used with a sample of data of much larger than the size of the partition. The expectation consequently uses a bootstrapping method to sample the provided data with tunable specificity.

- *expect_column_bootstrapped_ks_test_p_value_to_be_greater_than*

For categorical data, the expect_column_chisquare_test_p_value_to_be_greater_than expectation uses the Chi-Squared test. The Chi-Squared test works with expected and observed counts, but that is handled internally in this function – both the input and output to this function are valid partition objects (ie with weights that are probabilities and sum to 1).

- *expect_column_chisquare_test_p_value_to_be_greater_than*

Distributional Expectations Alternatives

The core partition density object used in current expectations focuses on a particular (partition-based) method of "compressing" the data into a testable form, however it may be desireable to use alternative nonparametric approaches (e.g. Fourier transform/wavelets) to describe expected data.

7.1.3 Building Custom Expectations

This document provides examples that walk through several methods for building and deploying custom expectations. Most of the core Great Expectations expectations are built using expectation decorators, and using decorators on existing logic can make bringing custom integrations into your pipeline tests easy.

Using Expectation Decorators

Under the hood, great_expectations evaluates similar kinds of expectations using standard logic, including:

- *column_map_expectations*, which apply their condition to each value in a column independently of other values
- *column_aggregate_expectations*, which apply their condition to an aggregate value or values from the column

In general, if a column is empty, a column_map_expectation will return True (vacuously), whereas a column_aggregate_expectation will return False (since no aggregate value could be computed).

Adding an expectation about element counts to a set of expectations is usually therefore very important to ensure the overall set of expectations captures the full set of constraints you expect.

High-level decorators

High-level decorators may modify the type of arguments passed to their decorated methods and do significant computation or bookkeeping to augment the work done in an expectation. That makes implementing many common types of operations using high-level decorators very easy.

To use the high-level decorators (e.g. `column_map_expectation` or `` `column_aggregate_expectation ``):

1. Create a subclass from the dataset class of your choice
2. Define custom functions containing your business logic
3. Use the *column_map_expectation* and *column_aggregate_expectation* decorators to turn them into full Expectations. Note that each dataset class implements its own versions of *@column_map_expectation* and *@column_aggregate_expectation*, so you should consult the documentation of each class to ensure you are returning the correct information to the decorator.

Note: following Great Expectations patterns for *Extending Great Expectations* is highly recommended, but not strictly required. If you want to confuse yourself with bad names, the package won't stop you.

For example, in Pandas:

@MetaPandasDataset.column_map_expectation decorates a custom function, wrapping it with all the business logic required to turn it into a fully-fledged Expectation. This spares you the hassle of defining logic to handle required arguments like *mostly* and *result_format*. Your custom function can focus exclusively on the business logic of passing or failing the expectation.

To work with these decorators, your custom function must accept two arguments: *self* and *column*. When your function is called, *column* will contain all the non-null values in the given column. Your function must return a series of boolean values in the same order, with the same index.

@MetaPandasDataset.column_aggregate_expectation accepts *self* and *column*. It must return a dictionary containing a boolean *success* value, and a nested dictionary called *result* which contains an *observed_value* argument.

Setting the _data_asset_type is not strictly necessary, but doing so allows GE to recognize that you have added expectations rather than simply added support for the same expectation suite on a different backend/compute environment.

```
from great_expectations.dataset import PandasDataset, MetaPandasDataset

class CustomPandasDataset(PandasDataset):

    _data_asset_type = "CustomPandasDataset"

    @MetaPandasDataset.column_map_expectation
    def expect_column_values_to_equal_2(self, column):
        return column.map(lambda x: x==2)

    @MetaPandasDataset.column_aggregate_expectation
    def expect_column_mode_to_equal_0(self, column):
        mode = column.mode[0]
        return {
            "success" : mode == 0,
            "result": {
                "observed_value": mode,
            }
        }
```

For SqlAlchemyDataset and SparkDFDataset, the decorators work slightly differently. See the respective Meta-class docstrings for more information; the example below shows SqlAlchemy implementations of the same custom expectations.

```
import sqlalchemy as sa
from great_expectations.dataset import SqlAlchemyDataset, MetaSqlAlchemyDataset

class CustomSqlAlchemyDataset(SqlAlchemyDataset):

    _data_asset_type = "CustomSqlAlchemyDataset"

    @MetaSqlAlchemyDataset.column_map_expectation
    def expect_column_values_to_equal_2(self, column):
        return (sa.column(column) == 2)

    @MetaSqlAlchemyDataset.column_aggregate_expectation
    def expect_column_mode_to_equal_0(self, column):
        mode_query = sa.select([
            sa.column(column).label('value'),
            sa.func.count(sa.column(column)).label('frequency')
        ]).select_from(self._table).group_by(sa.column(column)).order_by(sa.desc(sa.
↪column('frequency')))

        mode = self.engine.execute(mode_query).scalar()
        return {
            "success": mode == 0,
            "result": {
                "observed_value": mode,
            }
        }
```

Using the base Expectation decorator

When the high-level decorators do not provide sufficient granularity for controlling your expectation's behavior, you need to use the base expectation decorator, which will handle storing and retrieving your expectation in an expectation suite, and facilitate validation using your expectation. You will need to explicitly declare the parameters.

1. Create a subclass from the dataset class of your choice
2. Write the whole expectation yourself
3. Decorate it with the *@expectation* decorator, declaring the parameters you will use.

This is more complicated, since you have to handle all the logic of additional parameters and output formats. Pay special attention to proper formatting of *result_format*.

```
from great_expectations.data_asset import DataAsset
from great_expectations.dataset import PandasDataset

class CustomPandasDataset(PandasDataset):

    _data_asset_type = "CustomPandasDataset"

    @DataAsset.expectation(["column", "mostly"])
    def expect_column_values_to_equal_1(self, column, mostly=None):
        not_null = self[column].notnull()

        result = self[column][not_null] == 1
        unexpected_values = list(self[column][not_null][result==False])

        if mostly:
            #Prevent division-by-zero errors
            if len(not_null) == 0:
                return {
                    'success':True,
                    'unexpected_list':unexpected_values,
                    'unexpected_index_list':self.index[result],
                }

            percent_equaling_1 = float(sum(result))/len(not_null)
            return {
                "success" : percent_equaling_1 >= mostly,
                "unexpected_list" : unexpected_values[:20],
                "unexpected_index_list" : list(self.index[result==False])[:20],
            }
        else:
            return {
                "success" : len(unexpected_values) == 0,
                "unexpected_list" : unexpected_values[:20],
                "unexpected_index_list" : list(self.index[result==False])[:20],
            }
```

A similar implementation for SqlAlchemy would also import the base decorator:

```
import sqlalchemy as sa
from great_expectations.data_asset import DataAsset
from great_expectations.dataset import SqlAlchemyDataset

import numpy as np
import scipy.stats as stats
```

(continues on next page)

(continued from previous page)

```
import scipy.special as special

if sys.version_info.major >= 3 and sys.version_info.minor >= 5:
    from math import gcd
else:
    from fractions import gcd

class CustomSqlAlchemyDataset(SqlAlchemyDataset):

    _data_asset_type = "CustomSqlAlchemyDataset"

    @DataAsset.expectation(["column_A", "column_B", "p_value", "mode"])
    def expect_column_pair_histogram_ks_2samp_test_p_value_to_be_greater_than(
            self,
            column_A,
            column_B,
            p_value=0.05,
            mode='auto'
    ):
        """Execute the two sample KS test on two columns of data that are expected to␣
↪be **histograms** with
        aligned values/points on the CDF. ."""
        LARGE_N = 10000  # 'auto' will attempt to be exact if n1,n2 <= LARGE_N

        # We will assume that these are already HISTOGRAMS created as a check_dataset
        # either of binned values or of (ordered) value counts
        rows = sa.select([
            sa.column(column_A).label("col_A_counts"),
            sa.column(column_B).label("col_B_counts")
        ]).select_from(self._table).fetchall()

        cols = [col for col in zip(*rows)]
        cdf1 = np.array(cols[0])
        cdf2 = np.array(cols[1])
        n1 = cdf1.sum()
        n2 = cdf2.sum()
        cdf1 = cdf1 / n1
        cdf2 = cdf2 / n2

        # This code is taken verbatim from scipy implementation,
        # skipping the searchsorted (using sqlalchemy check asset as a view)
        # https://github.com/scipy/scipy/blob/v1.3.1/scipy/stats/stats.py#L5385-L5573
        cddiffs = cdf1 - cdf2
        minS = -np.min(cddiffs)
        maxS = np.max(cddiffs)
        alt2Dvalue = {'less': minS, 'greater': maxS, 'two-sided': max(minS, maxS)}
        d = alt2Dvalue[alternative]
        g = gcd(n1, n2)
        n1g = n1 // g
        n2g = n2 // g
        prob = -np.inf
        original_mode = mode
        if mode == 'auto':
            if max(n1, n2) <= LARGE_N:
                mode = 'exact'
            else:
                mode = 'asymp'
```

(continues on next page)

(continued from previous page)

```
        elif mode == 'exact':
            # If lcm(n1, n2) is too big, switch from exact to asymp
            if n1g >= np.iinfo(np.int).max / n2g:
                mode = 'asymp'
                warnings.warn(
                    "Exact ks_2samp calculation not possible with samples sizes "
                    "%d and %d. Switching to 'asymp' " % (n1, n2), RuntimeWarning)

        saw_fp_error = False
        if mode == 'exact':
            lcm = (n1 // g) * n2
            h = int(np.round(d * lcm))
            d = h * 1.0 / lcm
            if h == 0:
                prob = 1.0
            else:
                try:
                    if alternative == 'two-sided':
                        if n1 == n2:
                            prob = stats._compute_prob_outside_square(n1, h)
                        else:
                            prob = 1 - stats._compute_prob_inside_method(n1, n2, g, h)
                    else:
                        if n1 == n2:
                            # prob = binom(2n, n-h) / binom(2n, n)
                            # Evaluating in that form incurs roundoff errors
                            # from special.binom. Instead calculate directly
                            prob = 1.0
                            for j in range(h):
                                prob = (n1 - j) * prob / (n1 + j + 1)
                        else:
                            num_paths = stats._count_paths_outside_method(n1, n2, g,␣
↪h)
                            bin = special.binom(n1 + n2, n1)
                            if not np.isfinite(bin) or not np.isfinite(num_paths) or␣
↪num_paths > bin:
                                raise FloatingPointError()
                            prob = num_paths / bin

                except FloatingPointError:
                    # Switch mode
                    mode = 'asymp'
                    saw_fp_error = True
                    # Can't raise warning here, inside the try
                finally:
                    if saw_fp_error:
                        if original_mode == 'exact':
                            warnings.warn(
                                "ks_2samp: Exact calculation overflowed. "
                                "Switching to mode=%s" % mode, RuntimeWarning)
                    else:
                        if prob > 1 or prob < 0:
                            mode = 'asymp'
                            if original_mode == 'exact':
                                warnings.warn(
                                    "ks_2samp: Exact calculation incurred large"
                                    " rounding error. Switching to mode=%s" % mode,
```

(continues on next page)

(continued from previous page)

```
                                    RuntimeWarning)

        if mode == 'asymp':
            # The product n1*n2 is large.  Use Smirnov's asymptoptic formula.
            if alternative == 'two-sided':
                en = np.sqrt(n1 * n2 / (n1 + n2))
                # Switch to using kstwo.sf() when it becomes available.
                # prob = distributions.kstwo.sf(d, int(np.round(en)))
                prob = distributions.kstwobign.sf(en * d)
            else:
                m, n = max(n1, n2), min(n1, n2)
                z = np.sqrt(m*n/(m+n)) * d
                # Use Hodges' suggested approximation Eqn 5.3
                expt = -2 * z**2 - 2 * z * (m + 2*n)/np.sqrt(m*n*(m+n))/3.0
                prob = np.exp(expt)

        prob = (0 if prob < 0 else (1 if prob > 1 else prob))

        return {
            "success": prob > p_value,
            "result": {
                "observed_value": prob,
                "details": {
                    "ks_2samp_statistic": d
                }
            }
        }
```

Rapid Prototyping

For rapid prototyping, the following syntax allows quick iteration on the logic for expectations.

```
>> DataAsset.test_expectation_function(my_func)

>> Dataset.test_column_map_expectation_function(my_map_func, column='my_column')

>> Dataset.test_column_aggregate_expectation_function(my_agg_func, column='my_column')
```

These functions will return output just like regular expectations. However, they will NOT save a copy of the expectation to the current expectation suite.

Using custom expectations

Let's suppose you've defined *CustomPandasDataset* in a module called *custom_dataset.py*. You can instantiate a dataset with your custom expectations simply by adding *dataset_class=CustomPandasDataset* in *ge.read_csv*.

Once you do this, all the functionality of your new expectations will be available for uses.

```
>> import great_expectations as ge
>> from custom_dataset import CustomPandasDataset

>> my_df = ge.read_csv("my_data_file.csv", dataset_class=CustomPandasDataset)

>> my_df.expect_column_values_to_equal_1("all_twos")
```

(continues on next page)

(continued from previous page)

```
{
    "success": False,
    "unexpected_list": [2,2,2,2,2,2,2,2]
}
```

A similar approach works for the command-line tool.

```
>> great_expectations validation csv \
    my_data_file.csv \
    my_expectations.json \
    dataset_class=custom_dataset.CustomPandasDataset
```

Using custom expectations with a Datasource

To use custom expectations in a datasource or DataContext, you need to define the custom DataAsset in the datasource configuration or batch_kwargs for a specific batch. Following the same example above, let's suppose you've defined *CustomPandasDataset* in a module called *custom_dataset.py*. You can configure your datasource to return instances of your custom DataAsset type by declaring that as the data_asset_type for the datasource to build.

If you are working a DataContext, simply placing *custom_dataset.py* in your configured plugin directory will make it accessible, otherwise, you need to ensure the module is on the import path.

Once you do this, all the functionality of your new expectations will be available for use. For example, you could use the datasource snippet below to configure a PandasDatasource that will produce instances of your new CustomPandasDataset in a DataContext. Note the use of standard python dot notation to import.

```
datasources:
  my_datasource:
    class_name: PandasDatasource
    data_asset_type:
      module_name: custom_module.custom_dataset
      class_name: CustomPandasDataset
    generators:
      default:
        class_name: SubdirReaderBatchKwargsGenerator
        base_directory: /data
        reader_options:
          sep: \t
```

Note that we need to have added our **custom_dataset.py** to a directory called **custom_module** as in the directory structure below.

```
great_expectations
├── .gitignore
├── datasources
├── expectations
├── great_expectations.yml
├── notebooks
│   ├── pandas
│   ├── spark
│   └── sql
├── plugins
│   └── custom_module
│       └── custom_dataset.py
└── uncommitted
```

(continues on next page)

(continued from previous page)

```
    ├── config_variables.yml
    ├── data_docs
    │   └── local_site
    ├── samples
    └── validations
```

```
>> import great_expectations as ge
>> context = ge.DataContext()
>> my_df = context.get_batch(
    "my_datasource/default/my_file",
    "warning",
    context.yield_batch_kwargs("my_datasource/default/my_file"))

>> my_df.expect_column_values_to_equal_1("all_twos")
{
    "success": False,
    "unexpected_list": [2,2,2,2,2,2,2,2]
}
```

7.1.4 Table of Expectation Implementations By Backend

Because Great Expectations can run against different platforms, not all expectations have been implemented for all platforms. This table details which are implemented. Note we love pull-requests to help us fill out the missing implementations!

Expectations	Pandas	SQL	Spark
expect_column_to_exist	Y	Y	Y
expect_table_columns_to_match_ordered_list	Y	Y	Y
expect_table_row_count_to_be_between	Y	Y	Y
expect_table_row_count_to_equal	Y	Y	Y
expect_column_values_to_be_unique	Y	Y	Y
expect_column_values_to_not_be_null	Y	Y	Y
expect_column_values_to_be_null	Y	Y	Y
expect_column_values_to_be_of_type	Y	Y	Y
expect_column_values_to_be_in_type_list	Y	Y	Y
expect_column_values_to_be_in_set	Y	Y	Y
expect_column_values_to_not_be_in_set	Y	Y	Y
expect_column_values_to_be_between	Y	Y	Y
expect_column_values_to_be_increasing	Y	N	N
expect_column_values_to_be_decreasing	Y	N	N
expect_column_value_lengths_to_be_between	Y	Y	Y
expect_column_value_lengths_to_equal	Y	Y	Y
expect_column_values_to_match_regex	Y	Y	Y
expect_column_values_to_not_match_regex	Y	Y	Y
expect_column_values_to_match_regex_list	Y	Y	N
expect_column_values_to_not_match_regex_list	Y	Y	N
expect_column_values_to_match_strftime_format	Y	N	N
expect_column_values_to_be_dateutil_parseable	Y	N	N
expect_column_values_to_be_json_parseable	Y	N	N
expect_column_values_to_match_json_schema	Y	N	N

continues on next page

Table 1 – continued from previous page

expect_column_parameterized_distribution_ks_test_p_value_to_be_greater_than	Y	N	N
expect_column_distinct_values_to_equal_set	Y	Y	Y
expect_column_distinct_values_to_contain_set	Y	Y	Y
expect_column_mean_to_be_between	Y	Y	Y
expect_column_median_to_be_between	Y	Y	Y
expect_column_stdev_to_be_between	Y	N	Y
expect_column_unique_value_count_to_be_between	Y	Y	Y
expect_column_proportion_of_unique_values_to_be_between	Y	Y	Y
expect_column_most_common_value_to_be_in_set	Y	N	Y
expect_column_sum_to_be_between	Y	Y	Y
expect_column_min_to_be_between	Y	Y	Y
expect_column_max_to_be_between	Y	Y	Y
expect_column_chisquare_test_p_value_to_be_greater_than	Y	Y	Y
expect_column_bootstrapped_ks_test_p_value_to_be_greater_than	Y	N	N
expect_column_kl_divergence_to_be_less_than	Y	Y	Y
expect_column_pair_values_to_be_equal	Y	N	Y
expect_column_pair_values_A_to_be_greater_than_B	Y	N	N
expect_column_pair_values_to_be_in_set	Y	N	N
expect_multicolumn_values_to_be_unique	Y	N	N

7.2 Core GE Types

7.2.1 Standard arguments for expectations

All Expectations return a json-serializable dictionary when evaluated, and share four standard (optional) arguments:

- *result_format*: controls what information is returned from the evaluation of the expectation expectation.
- *include_config*: If true, then the expectation suite itself is returned as part of the result object.
- *catch_exceptions*: If true, execution will not fail if the Expectation encounters an error. Instead, it will return success = False and provide an informative error message.
- *meta*: allows user-supplied meta-data to be stored with an expectation.

result_format

See *result_format* for more information.

include_config

All Expectations accept a boolean *include_config* parameter. If true, then the expectation suite itself is returned as part of the result object

```
>> expect_column_values_to_be_in_set(
    "my_var",
    ['B', 'C', 'D', 'F', 'G', 'H'],
    result_format="COMPLETE",
    include_config=True,
)
```

(continues on next page)

(continued from previous page)

```
{
    'exception_index_list': [0, 10, 11, 12, 13, 14],
    'exception_list': ['A', 'E', 'E', 'E', 'E', 'E'],
    'expectation_type': 'expect_column_values_to_be_in_set',
    'expectation_kwargs': {
        'column': 'my_var',
        'result_format': 'COMPLETE',
        'value_set': ['B', 'C', 'D', 'F', 'G', 'H']
    },
    'success': False
}
```

catch_exceptions

All Expectations accept a boolean *catch_exceptions* parameter. If true, execution will not fail if the Expectation encounters an error. Instead, it will return False and (in *BASIC* and *SUMMARY* modes) an informative error message

```
{
    "result": False,
    "raised_exception": True,
    "exception_traceback": "..."
}
```

catch_exceptions is on by default in command-line validation mode, and off by default in exploration mode.

meta

All Expectations accept an optional *meta* parameter. If *meta* is a valid JSON-serializable dictionary, it will be passed through to the *expectation_result* object without modification. The *meta* parameter can be used to add helpful markdown annotations to Expectations (shown below). These Expectation "notes" are rendered within Expectation Suite pages in Data Docs.

```
>> my_df.expect_column_values_to_be_in_set(
    "my_column",
    ["a", "b", "c"],
    meta={
      "notes": {
        "format": "markdown",
        "content": [
          "#### These are expectation notes \n - you can use markdown \n - or just␣
↪strings"
        ]
      }
    }
)
{
    "success": False,
    "meta": {
      "notes": {
        "format": "markdown",
        "content": [
          "#### These are expectation notes \n - you can use markdown \n - or just␣
↪strings"
        ]
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

mostly

mostly is a special argument that is automatically available in all *column_map_expectations*. *mostly* must be a float between 0 and 1. Great Expectations evaluates it as a percentage, allowing some wiggle room when evaluating expectations: as long as *mostly* percent of rows evaluate to *True*, the expectation returns *"success": True*.

```
[0,1,2,3,4,5,6,7,8,9]

>> my_df.expect_column_values_to_be_between(
    "my_column",
    min_value=0,
    max_value=7
)
{
    "success": False,
    ...
}

>> my_df.expect_column_values_to_be_between(
    "my_column",
    min_value=0,
    max_value=7,
    mostly=0.7
)
{
    "success": True,
    ...
}
```

Expectations with *mostly* return exception lists even if they succeed:

```
>> my_df.expect_column_values_to_be_between(
    "my_column",
    min_value=0,
    max_value=7,
    mostly=0.7
)
{
  "success": true
  "result": {
    "unexpected_percent": 0.2,
    "partial_unexpected_index_list": [
      8,
      9
    ],
    "partial_unexpected_list": [
      8,
      9
    ],
    "unexpected_percent_nonmissing": 0.2,
    "unexpected_count": 2
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Dataset defaults

This default behavior for *result_format*, *include_config*, *catch_exceptions* can be overridden at the Dataset level:

```
my_dataset.set_default_expectation_argument("result_format", "SUMMARY")
```

In validation mode, they can be overridden using flags:

```
great_expectations validation csv my_dataset.csv my_expectations.json --result_
↪format=BOOLEAN_ONLY --catch_exceptions=False --include_config=True
```

7.2.2 result_format

The *result_format* parameter may be either a string or a dictionary which specifies the fields to return in *result*.

- For string usage, see *result_format*.
- For dictionary usage, *result_format* which may include the following keys:
  - *result_format*: Sets the fields to return in result.
  - partial_unexpected_count: Sets the number of results to include in partial_unexpected_count, if applicable. If set to 0, this will suppress the unexpected counts.

result_format

Great Expectations supports four values for *result_format*: *BOOLEAN_ONLY*, *BASIC*, *SUMMARY*, and *COMPLETE*. Each successive value includes more detail and so can support different use cases for working with Great Expectations, including interactive exploratory work and automatic validation.

Fields within *result*	BOOLEAN_ONLY	BASIC	SUMMARY	COMPLETE
element_count	no	yes	yes	yes
missing_count	no	yes	yes	yes
missing_percent	no	yes	yes	yes
details (dictionary)	Defined on a per-expectation basis			
Fields defined for *column_map_expectation* type expectations:				
unexpected_count	no	yes	yes	yes
unexpected_percent	no	yes	yes	yes
unexpected_percent_nonmissing	no	yes	yes	yes
partial_unexpected_list	no	yes	yes	yes
partial_unexpected_index_list	no	no	yes	yes
partial_unexpected_counts	no	no	yes	yes
unexpected_index_list	no	no	no	yes
unexpected_list	no	no	no	yes
Fields defined for *column_aggregate_expectation* type expectations:				
observed_value	no	yes	yes	yes
details (e.g. statistical details)	no	no	yes	yes

result_format Setting	Example use case
BOOLEAN_ONLY	Automatic validation. No result is returned.
BASIC	Exploratory analysis in a notebook.
SUMMARY	Detailed exploratory work with follow-on investigation.
COMPLETE	Debugging pipelines or developing detailed regression tests.

result_format examples

```
>> print(list(my_df.my_var))
['A', 'B', 'B', 'C', 'C', 'C', 'D', 'D', 'D', 'D', 'E', 'E', 'E', 'E', 'E', 'F', 'F',
↪'F', 'F', 'F', 'F', 'G', 'G', 'G', 'G', 'G', 'G', 'G', 'H', 'H', 'H', 'H', 'H', 'H',
↪ 'H', 'H']

>> my_df.expect_column_values_to_be_in_set(
    "my_var",
    ["B", "C", "D", "F", "G", "H"],
    result_format={'result_format': 'BOOLEAN_ONLY'}
)
{
    'success': False
}

>> my_df.expect_column_values_to_be_in_set(
    "my_var",
    ["B", "C", "D", "F", "G", "H"],
    result_format={'result_format': 'BASIC'}
)
{
    'success': False,
    'result': {
        'unexpected_count': 6,
        'unexpected_percent': 0.1666666666666666,
        'unexpected_percent_nonmissing': 0.1666666666666666,
        'partial_unexpected_list': ['A', 'E', 'E', 'E', 'E', 'E']
    }
}

>> expect_column_values_to_match_regex(
    "my_column",
    "[A-Z][a-z]+",
    result_format={'result_format': 'SUMMARY'}
)
{
    'success': False,
    'result': {
        'element_count': 36,
        'unexpected_count': 6,
        'unexpected_percent': 0.1666666666666666,
        'unexpected_percent_nonmissing': 0.1666666666666666,
        'missing_count': 0,
        'missing_percent': 0.0,
        'partial_unexpected_counts': [{'value': 'A', 'count': 1}, {'value': 'E',
↪'count': 5}],
        'partial_unexpected_index_list': [0, 10, 11, 12, 13, 14],
        'partial_unexpected_list': ['A', 'E', 'E', 'E', 'E', 'E']
```

(continues on next page)

(continued from previous page)

```
    }
}

>> my_df.expect_column_values_to_be_in_set(
    "my_var",
    ["B", "C", "D", "F", "G", "H"],
    result_format={'result_format': 'COMPLETE'}
)
{
    'success': False,
    'result': {
        'unexpected_index_list': [0, 10, 11, 12, 13, 14],
        'unexpected_list': ['A', 'E', 'E', 'E', 'E', 'E']
    }
}
```

The out-of-the-box default is *{'result_format'='BASIC'}*.

Behavior for *BOOLEAN_ONLY*

When the *result_format* is *BOOLEAN_ONLY*, no *result* is returned. The result of evaluating the expectation is exclusively returned via the value of the *success* parameter.

For example:

```
>> my_df.expect_column_values_to_be_in_set(
    "possible_benefactors",
    ["Joe Gargery", "Mrs. Gargery", "Mr. Pumblechook", "Ms. Havisham", "Mr. Jaggers"]
    result_format={'result_format': 'BOOLEAN_ONLY'}
)
{
    'success': False
}

>> my_df.expect_column_values_to_be_in_set(
    "possible_benefactors",
    ["Joe Gargery", "Mrs. Gargery", "Mr. Pumblechook", "Ms. Havisham", "Mr. Jaggers",
↪"Mr. Magwitch"]
    result_format={'result_format': 'BOOLEAN_ONLY'}
)
{
    'success': False
}
```

Behavior for *BASIC*

A *result* is generated with a basic justification for why an expectation was met or not. The format is intended for quick, at-a-glance feedback. For example, it tends to work well in jupyter notebooks.

Great Expectations has standard behavior for support for describing the results of *column_map_expectation* and *column_aggregate_expectation* expectations.

column_map_expectation applies a boolean test function to each element within a column, and so returns a list of unexpected values to justify the expectation result.

The basic *result* includes:

```
{
    "success" : Boolean,
    "result" : {
        "partial_unexpected_list" : [A list of up to 20 values that violate the␣
↪expectation]
        "unexpected_count" : The total count of unexpected values in the column
        "unexpected_percent" : The overall percent of unexpected values
        "unexpected_percent_nonmissing" : The percent of unexpected values, excluding␣
↪missing values from the denominator
    }
}
```

Note: when unexpected values are duplicated, *unexpected_list* will contain multiple copies of the value.

```
[1,2,2,3,3,3,None,None,None,None]

expect_column_values_to_be_unique

{
    "success" : Boolean,
    "result" : {
        "partial_unexpected_list" : [2,2,3,3,3]
        "unexpected_count" : 5,
        "unexpected_percent" : 0.5,
        "unexpected_percent_nonmissing" : 0.8333333
    }
}
```

column_aggregate_expectation computes a single aggregate value for the column, and so returns a single *observed_value* to justify the expectation result.

The basic *result* includes:

```
{
    "success" : Boolean,
    "result" : {
        "observed_value" : The aggregate statistic computed for the column
    }
}
```

For example:

```
[1, 1, 2, 2]

expect_column_mean_to_be_between

{
    "success" : Boolean,
    "result" : {
        "observed_value" : 1.5
    }
}
```

Behavior for *SUMMARY*

A *result* is generated with a summary justification for why an expectation was met or not. The format is intended for more detailed exploratory work and includes additional information beyond what is included by *BASIC*. For example, it can support generating dashboard results of whether a set of expectations are being met.

Great Expectations has standard behavior for support for describing the results of *column_map_expectation* and *column_aggregate_expectation* expectations.

column_map_expectation applies a boolean test function to each element within a column, and so returns a list of unexpected values to justify the expectation result.

The summary *result* includes:

```
{
    'success': False,
    'result': {
        'element_count': The total number of values in the column
        'unexpected_count': The total count of unexpected values in the column (also␣
↪in `BASIC`)
        'unexpected_percent': The overall percent of unexpected values (also in␣
↪`BASIC`)
        'unexpected_percent_nonmissing': The percent of unexpected values, excluding␣
↪missing values from the denominator (also in `BASIC`)
        "partial_unexpected_list" : [A list of up to 20 values that violate the␣
↪expectation] (also in `BASIC`)
        'missing_count': The number of missing values in the column
        'missing_percent': The total percent of missing values in the column
        'partial_unexpected_counts': [{A list of objects with value and counts,␣
↪showing the number of times each of the unexpected values occurs}]
        'partial_unexpected_index_list': [A list of up to 20 of the indices of the␣
↪unexpected values in the column]
    }
}
```

For example:

```
{
    'success': False,
    'result': {
        'element_count': 36,
        'unexpected_count': 6,
        'unexpected_percent': 0.1666666666666666,
        'unexpected_percent_nonmissing': 0.1666666666666666,
        'missing_count': 0,
        'missing_percent': 0.0,
        'partial_unexpected_counts': [{'value': 'A', 'count': 1}, {'value': 'E',
↪'count': 5}],
        'partial_unexpected_index_list': [0, 10, 11, 12, 13, 14],
        'partial_unexpected_list': ['A', 'E', 'E', 'E', 'E', 'E']
    }
}
```

column_aggregate_expectation computes a single aggregate value for the column, and so returns a *observed_value* to justify the expectation result. It also includes additional information regarding observed values and counts, depending on the specific expectation.

The summary *result* includes:

```
{
    'success': False,
    'result': {
        'observed_value': The aggregate statistic computed for the column (also in␣
↪`BASIC`)
        'element_count': The total number of values in the column
        'missing_count':  The number of missing values in the column
        'missing_percent': The total percent of missing values in the column
        'details': {<expectation-specific result justification fields>}
    }
}
```

For example:

```
[1, 1, 2, 2, NaN]

expect_column_mean_to_be_between

{
    "success" : Boolean,
    "result" : {
        "observed_value" : 1.5,
        'element_count': 5,
        'missing_count: 1,
        'missing_percent: 0.2
    }
}
```

Behavior for *COMPLETE*

A *result* is generated with all available justification for why an expectation was met or not. The format is intended for debugging pipelines or developing detailed regression tests.

Great Expectations has standard behavior for support for describing the results of *column_map_expectation* and *column_aggregate_expectation* expectations.

column_map_expectation applies a boolean test function to each element within a column, and so returns a list of unexpected values to justify the expectation result.

The complete *result* includes:

```
{
    'success': False,
    'result': {
        "unexpected_list" : [A list of all values that violate the expectation]
        'unexpected_index_list': [A list of the indices of the unexpected values in␣
↪the column]
        'element_count': The total number of values in the column (also in `SUMMARY`)
        'unexpected_count': The total count of unexpected values in the column (also␣
↪in `SUMMARY`)
        'unexpected_percent': The overall percent of unexpected values (also in␣
↪`SUMMARY`)
        'unexpected_percent_nonmissing': The percent of unexpected values, excluding␣
↪missing values from the denominator (also in `SUMMARY`)
        'missing_count': The number of missing values in the column  (also in␣
↪`SUMMARY`)
        'missing_percent': The total percent of missing values in the column  (also␣
↪in `SUMMARY`)
```

(continues on next page)

(continued from previous page)

```
    }
}
```

For example:

```
{
    'success': False,
    'result': {
        'element_count': 36,
        'unexpected_count': 6,
        'unexpected_percent': 0.16666666666666666,
        'unexpected_percent_nonmissing': 0.16666666666666666,
        'missing_count': 0,
        'missing_percent': 0.0,
        'unexpected_index_list': [0, 10, 11, 12, 13, 14],
        'unexpected_list': ['A', 'E', 'E', 'E', 'E', 'E']
    }
}
```

column_aggregate_expectation computes a single aggregate value for the column, and so returns a *observed_value* to justify the expectation result. It also includes additional information regarding observed values and counts, depending on the specific expectation.

The complete *result* includes:

```
{
    'success': False,
    'result': {
        'observed_value': The aggregate statistic computed for the column (also in␣
↪`SUMMARY`)
        'element_count': The total number of values in the column (also in `SUMMARY`)
        'missing_count':  The number of missing values in the column (also in␣
↪`SUMMARY`)
        'missing_percent': The total percent of missing values in the column (also in␣
↪`SUMMARY`)
        'details': {<expectation-specific result justification fields, which may be␣
↪more detailed than in `SUMMARY`>}
    }
}
```

For example:

```
[1, 1, 2, 2, NaN]

expect_column_mean_to_be_between

{
    "success" : Boolean,
    "result" : {
        "observed_value" : 1.5,
        'element_count': 5,
        'missing_count: 1,
        'missing_percent: 0.2
    }
}
```

7.2.3 Validation Results

Validation results include information about each expectation that was validated and a `statistics` section that describes the overall number of validated expectations and results.

For example:


```
{
  "meta": {
    "data_asset_name": "data__dir/default/titanic",
    "expectation_suite_name": "default",
    "run_id": "2019-07-01T123434.12345Z"
  },
  "results": [
    {
      "expectation_config": {
        "expectation_type": "expect_column_values_to_have_odd_lengths",
        "kwargs": {
          "column": "Name",
          "result_format": "SUMMARY"
        }
      },
      "exception_info": {
        "exception_message": null,
        "exception_traceback": null,
        "raised_exception": false
      },
      "success": false,
      "result": {
        "partial_unexpected_index_list": [
          0,
          5,
          6,
          7,
          8,
          9,
          11,
          12,
          13,
          14,
          15,
          18,
          20,
          21,
          22,
          23,
          27,
          31,
          32,
          33
        ],
        "unexpected_count": 660,
        "unexpected_percent": 0.5026656511805027,
        "partial_unexpected_list": [
          "Allen, Miss Elisabeth Walton",
          "Anderson, Mr Harry",
          "Andrews, Miss Kornelia Theodosia",
          "Andrews, Mr Thomas, jr",
```


(continues on next page)

(continued from previous page)


```
            "Appleton, Mrs Edward Dale (Charlotte Lamson)",
            "Artagaveytia, Mr Ramon",
            "Astor, Mrs John Jacob (Madeleine Talmadge Force)",
            "Aubert, Mrs Leontine Pauline",
            "Barkworth, Mr Algernon H",
            "Baumann, Mr John D",
            "Baxter, Mrs James (Helene DeLaudeniere Chaput)",
            "Beckwith, Mr Richard Leonard",
            "Behr, Mr Karl Howell",
            "Birnbaum, Mr Jakob",
            "Bishop, Mr Dickinson H",
            "Bishop, Mrs Dickinson H (Helen Walton)",
            "Bonnell, Miss Caroline",
            "Bowerman, Miss Elsie Edith",
            "Bradley, Mr George",
            "Brady, Mr John Bertram"
          ],
          "missing_percent": 0.0,
          "element_count": 1313,
          "unexpected_percent_nonmissing": 0.5026656511805027,
          "missing_count": 0
        }
      }
    ],
    "success": false,
    "statistics": {
      "evaluated_expectations": 1,
      "successful_expectations": 0,
      "unsuccessful_expectations": 1,
      "success_percent": 0
    }
}
```


Reviewing Validation Results

TODO: Description of process for reviewing validation results

7.2.4 Datasource Reference

To have a Datasource produce Data Assets of a custom type, such as when adding custom expectations by subclassing an existing DataAsset type, use the *data_asset_type* parameter to configure the datasource to load and return DataAssets of the custom type.

For example:

```
datasources:
  pandas:
    class_name: PandasDatasource
    data_asset_type:
      class_name: MyCustomPandasAsset
      module_name: internal_pandas_assets
```

Given the above configuration, we can observe the following:

```
>>> batch_kwargs = {
...    "datasource": "pandas",
...    "dataset": {"a": [1, 2, 3]}
... }
>>> batch = context.get_batch(batch_kwargs, my_suite)
>>> isinstance(batch, MyCustomPandasAsset)
True
```

Last updated: |lastupdate|

7.2.5 Batch Kwargs

Batch Kwargs represent the information required by a *Datasources* to fetch a batch of data.

The *partition_id* provides a single string that can be used to represent a data asset inside the namespace defined by a given datasource/generator/generator_asset triple.

7.3 Configuration Guides

7.3.1 Data Context Reference

A *DataContexts* manages assets for a project.

Configuration

The DataContext configuration file (`great_expectations.yml`) provides fine-grained control over several core features available to the DataContext to assist in managing resources used by Great Expectations. Key configuration areas include specifying Datasources, Data Docs, Validation Operators, and Stores used to manage access to resources such as expectation suites, validation results, profiling results, evaluation parameters and plugins.

This file is intended to be committed to your repo.

Datasources

Datasources tell Great Expectations where your data lives and how to get it.

Using the *CLI* command `great_expectations datasource new` is the easiest way to add a new datasource.

The *datasources* section declares which *Datasources* objects should be available in the DataContext. Each datasource definition should include the *class_name* of the datasource, generators, and any other relevant configuration information. For example, the following simple configuration supports a Pandas-based pipeline:

```
datasources:
  pipeline:
    class_name: PandasDatasource
    generators:
      default:
        class_name: InMemoryGenerator
```

The following configuration demonstrates a more complicated configuration for reading assets from s3 into pandas. It will access the amazon public NYC taxi data and provides access to two assets: 'taxi-green' and 'taxi-fhv' which represent two public datasets available from the resource.

```
datasources:
  nyc_taxi:
    class_name: PandasDatasource
    generators:
      s3:
        class_name: S3GlobReaderBatchKwargsGenerator
        bucket: nyc-tlc
        delimiter: '/'
        reader_options:
          sep: ','
          engine: python
        assets:
          taxi-green:
            prefix: trip data/
            regex_filter: 'trip data/green.*\.csv'
          taxi-fhv:
            prefix: trip data/
            regex_filter: 'trip data/fhv.*\.csv'
    data_asset_type:
      class_name: PandasDataset
```

Here is an example for a SQL based pipeline:

```
datasources:
  edw:
    class_name: SqlAlchemyDatasource
    credentials: ${data_warehouse}
    data_asset_type:
      class_name: SqlAlchemyDataset
    generators:
      default:
        class_name: TableBatchKwargsGenerator
```

Note the `credentials` key references a corresponding key in the `config_variables.yml` file which is not in source control that would look like this:

```
data_warehouse:
  drivername: postgres
  host: warehouse.ourcompany.biz
  port: '5432'
  username: bob
  password: 1234
  database: prod
```

Note that the datasources section *includes* all defined generators as well as specifying their names. See *Using custom expectations with a Datasource* for more information about configuring datasources to use custom expectations.

Data Asset Names

Data asset names consist of three parts, a datasource, generator, and generator asset. DataContext functions will attempt to "normalize" a data_asset_name if they are provided with only a string, by splitting on the delimiter character (by default '/') and then attempting to identify an unambiguous name. DataContext searches through names that already have expectation suites first, then considers names provided by generators.

For example:

```
# Returns a normalized name with string representation my_datasource/my_generator/my_
↪asset if
# my_datasource and my_generator uniquely provide an asset called my_asset
context.normalize_data_asset_name("my_asset")
```

Data Docs

The *Data Docs* section defines how individual sites should be built and deployed. See the detailed documentation for more information.

Stores

A DataContext requires three *stores* to function properly: an *expectations_store*, *validations_store*, and *evaluation_parameter_store*. Consequently a minimal store configuration for a DataContext would include the following:

```
expectations_store_name: expectations_store
validations_store_name: validations_store
evaluation_parameter_store_name: evaluation_parameter_store

stores:
  expectations_store:
    class_name: ExpectationsStore
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: expectations/
  validations_store:
    class_name: ValidationsStore
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: uncommitted/validations/
  evaluation_parameter_store:
    class_name: EvaluationParameterStore
```

The *expectations_store* provides access to expectations_suite objects, using the DataContext's namespace; the *validations_store* does the same for validations. See *Evaluation Parameters* for more information on the evaluation parameters store.

Stores can be referenced in other objects in the DataContext. They provide a common API for accessing data independently of the backend where it is stored. For example, on a team that uses S3 to store expectation suites and validation results, updating the configuration to use cloud storage requires only changing the store class_name and providing the bucket/prefix combination:

```
expectations_store_name: expectations_store
validations_store_name: validations_store
evaluation_parameter_store_name: evaluation_parameter_store
```

(continues on next page)

(continued from previous page)

```
stores:
  expectations_store:
    class_name: ExpectationsStore
    store_backend:
      class_name: TupleS3StoreBackend
      base_directory: expectations/
      bucket: ge.my_org.com
      prefix:
  validations_store:
    class_name: ValidationsStore
    store_backend:
      class_name: TupleS3StoreBackend
      bucket: ge.my_org.com
      prefix: common_validations
  evaluation_parameter_store:
    class_name: EvaluationParameterStore
```

GE uses boto3 to access AWS, so credentials simply need to be available in any standard place searched by that library. You may also specify keyword arguments for boto3 to use in the *boto3_options key* of the store_backend configuration.

Validation Operators

See the *Validation Operators* for more information regarding configuring and using validation operators.

Managing Environment and Secrets

In a DataContext configuration, values that should come from the runtime environment or secrets can be injected via a separate config file or using environment variables. Use the `${var}` syntax in a config file to specify a variable to be substituted.

Config Variables File

DataContext accepts a parameter called `config_variables_file_path` which can include a file path from which variables to substitute should be read. The file needs to define top-level keys which are available to substitute into a DataContext configuration file. Keys from the config variables file can be defined to represent complex types such as a dictionary or list, which is often useful for configuring database access.

Variable substitution enables: 1) keeping secrets out of source control & 2) environment-based configuration changes such as staging vs prod.

When GE encounters substitution syntax (like `my_key: ${my_value}` or `my_key: $my_value`) in the config file it will attempt to replace the value of `my_key` with the value from an environment variable `my_value` or a corresponding key read from the file specified using `config_variables_file_path`.

```
prod_credentials:
  type: postgresql
  host: secure_server
  port: 5432
  username: username
  password: sensitive_password
  database: ge
```

(continues on next page)

(continued from previous page)

```
dev_credentials:
  type: postgresql
  host: localhost
  port: 5432
  username: dev
  password: dev
  database: ge
```

If the substitution value comes from the config variables file, it can be a simple (non-nested) value or a nested value such as a dictionary. If it comes from an environment variable, it must be a simple value.

Environment Variable Substitution

Environment variables will be substituted into a DataContext config with higher priority than values from the config variables file.

Default Out of Box Config File

Should you need a clean config file you can run `great_expectation init` in a new directory or use this template:

```
# Welcome to Great Expectations! Always know what to expect from your data.
#
# Here you can define datasources, batch kwarg generators, integrations and
# more. This file is intended to be committed to your repo. For help with
# configuration please:
#   - Read our docs: https://docs.greatexpectations.io/en/latest/reference/data_
↪context_reference.html#configuration
#   - Join our slack channel: http://greatexpectations.io/slack

config_version: 1

# Datasources tell Great Expectations where your data lives and how to get it.
# You can use the CLI command `great_expectations datasource new` to help you
# add a new datasource. Read more at https://docs.greatexpectations.io/en/latest/
↪features/datasource.html
datasources: {}
  edw:
    class_name: SqlAlchemyDatasource
    credentials: ${edw}
    data_asset_type:
      class_name: SqlAlchemyDataset
    generators:
      default:
        class_name: TableBatchKwargsGenerator

# This config file supports variable substitution which enables: 1) keeping
# secrets out of source control & 2) environment-based configuration changes
# such as staging vs prod.
#
# When GE encounters substitution syntax (like `my_key: ${my_value}` or
# `my_key: $my_value`) in the config file it will attempt to replace the value
# of `my_key` with the value from an environment variable `my_value` or a
```

(continues on next page)

(continued from previous page)

```
# corresponding key read from the file specified using
# `config_variables_file_path`. Environment variables take precedence.
#
# If the substitution value comes from the config variables file, it can be a
# simple (non-nested) value or a nested value such as a dictionary. If it comes
# from an environment variable, it must be a simple value. Read more at:
# https://docs.greatexpectations.io/en/latest/reference/data_context_reference.html
↪#managing-environment-and-secrets
config_variables_file_path: uncommitted/config_variables.yml

# The plugins_directory will be added to your python path for custom modules
# used to override and extend Great Expectations.
plugins_directory: plugins/

# Validation Operators are customizable workflows that bundle the validation of
# one or more expectation suites and subsequent actions. The example below
# stores validations and send a slack notification. To read more about
# customizing and extending these, read: https://docs.greatexpectations.io/en/latest/
↪features/validation_operators_and_actions.html
validation_operators:
  action_list_operator:
    # To learn how to configure sending Slack notifications during evaluation
    # (and other customizations), read: https://docs.greatexpectations.io/en/latest/
↪reference/validation_operators/perform_action_list_validation_operator.html
    class_name: ActionListValidationOperator
    action_list:
      - name: store_validation_result
        action:
          class_name: StoreValidationResultAction
      - name: store_evaluation_params
        action:
          class_name: StoreEvaluationParametersAction
      - name: update_data_docs
        action:
          class_name: UpdateDataDocsAction
      - name: send_slack_notification_on_validation_result
        action:
          class_name: SlackNotificationAction
          slack_webhook: ${validation_notification_slack_webhook}
          notify_on: all
          renderer:
            module_name: great_expectations.render.renderer.slack_renderer
            class_name: SlackRenderer
stores:
# Stores are configurable places to store things like Expectations, Validations
# Data Docs, and more. These are for advanced users only - most users can simply
# leave this section alone.
#
# Three stores are required: expectations, validations, and
# evaluation_parameters, and must exist with a valid store entry. Additional
# stores can be configured for uses such as data_docs, validation_operators, etc.
  expectations_store:
    class_name: ExpectationsStore
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: expectations/
  validations_store:
```

(continues on next page)

(continued from previous page)

```
    class_name: ValidationsStore
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: uncommitted/validations/
  evaluation_parameter_store:
    # Evaluation Parameters enable dynamic expectations. Read more here:
    # https://docs.greatexpectations.io/en/latest/reference/evaluation_parameters.html
    class_name: EvaluationParameterStore
expectations_store_name: expectations_store
validations_store_name: validations_store
evaluation_parameter_store_name: evaluation_parameter_store

data_docs_sites:
  # Data Docs make it simple to visualize data quality in your project. These
  # include Expectations, Validations & Profiles. The are built for all
  # Datasources from JSON artifacts in the local repo including validations &
  # profiles from the uncommitted directory. Read more at https://docs.
↪greatexpectations.io/en/latest/features/data_docs.html
  local_site:
    class_name: SiteBuilder
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: uncommitted/data_docs/local_site/
```

7.3.2 Data Docs Reference

Data Docs make it simple to visualize data quality in your project. These include Expectations, Validations & Profiles. They are built for all Datasources from JSON artifacts in the local repo including validations & profiles from the uncommitted directory.

Users have full control over configuring Data Documentation for their project - they can modify the pre-configured site (or remove it altogether) and add new sites with a configuration that meets the project's needs. The easiest way to add a new site to the configuration is to copy the "local_site" configuration block in great_expectations.yml, give the copy a new name and modify the details as needed.

Data Docs Site Configuration

The default Data Docs site configuration looks like this:

```
data_docs_sites:
  local_site:
    class_name: SiteBuilder
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: uncommitted/data_docs/local_site/
    site_index_builder:
      class_name: DefaultSiteIndexBuilder
```

Here is an example of a site configuration from great_expectations.yml with defaults defined explicitly:

```
data_docs_sites:
  local_site: # site name
    module_name: great_expectations.render.renderer.site_builder
    class_name: SiteBuilder
```

(continues on next page)

(continued from previous page)

```
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: uncommitted/data_docs/local_site/
    site_index_builder:
      class_name: DefaultSiteIndexBuilder
    site_section_builders:
      expectations:  # if empty, or one of ['0', 'None', 'False', 'false', 'FALSE',
↪'none', 'NONE'], section not rendered
        class_name: DefaultSiteSectionBuilder
        source_store_name: expectations_store
        renderer:
          module_name: great_expectations.render.renderer
          class_name: ExpectationSuitePageRenderer

      validations:  # if empty, or one of ['0', 'None', 'False', 'false', 'FALSE',
↪'none', 'NONE'], section not rendered
        class_name: DefaultSiteSectionBuilder
        source_store_name: validations_store
        run_id_filter:
          ne: profiling # exclude validations with run id "profiling" - reserved for␣
↪profiling results
        renderer:
          module_name: great_expectations.render.renderer
          class_name: ValidationResultsPageRenderer

      profiling:  # if empty, or one of ['0', 'None', 'False', 'false', 'FALSE', 'none
↪', 'NONE'], section not rendered
        class_name: DefaultSiteSectionBuilder
        source_store_name: validations_store
        run_id_filter:
          eq: profiling
        renderer:
          module_name: great_expectations.render.renderer
          class_name: ProfilingResultsPageRenderer
```

`validations_store` in the example above specifies the name of a store configured in the *stores* section. Validation and profiling results from that store will be included in the documentation. The optional `run_id_filter` attribute allows to include (`eq` for exact match) or exclude (`ne`) validation results with a particular run id.

Limiting Validation Results

If you would like to limit rendered Validation Results to the n most-recent, you may do so by setting the *validation_results_limit* key in your Data Docs configuration:

```
data_docs_sites:
  local_site:
    class_name: SiteBuilder
    show_how_to_buttons: true
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: uncommitted/data_docs/local_site/
    site_index_builder:
      class_name: DefaultSiteIndexBuilder
      validation_results_limit: 5
```

Automatically Publishing Data Docs

It is possible to directly publish (continuously updating!) data docs sites to a shared location such as a static site hosted in S3 by simply updating the store_backend configuration in the site configuration. If we modify the configuration in the example above to adjust the store backend to an S3 bucket of our choosing, our *SiteBuilder* will automatically save the resulting site to that bucket.

```
store_backend:
  class_name: TupleS3StoreBackend
  bucket: data-docs.my_org.org
  prefix:
```

See the tutorial on *publishing data docs to S3* for more information.

More advanced configuration

It is possible to extend renderers and views and customize the particular class used to render any of the objects in your documentation. In this more advanced configuration, a "CustomTableContentBlockRenderer" is used only for the validations renderer, and no profiling results are rendered at all.

```
data_docs_sites:
  # Data Docs make it simple to visualize data quality in your project. These
  # include Expectations, Validations & Profiles. The are built for all
  # Datasources from JSON artifacts in the local repo including validations &
  # profiles from the uncommitted directory. Read more at
  # https://docs.greatexpectations.io/en/latestfeatures/data_docs.html
  local_site:
    class_name: SiteBuilder
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: uncommitted/data_docs/local_site/
    site_section_builders:
      validations:  # if empty, or one of ['0', 'None', 'False', 'false', 'FALSE',
↪'none', 'NONE'], section not rendered
        class_name: DefaultSiteSectionBuilder
        source_store_name: validations_store
        run_id_filter:
          ne: profiling
        renderer:
          module_name: great_expectations.render.renderer
          class_name: ValidationResultsPageRenderer
          column_section_renderer:
            class_name: ValidationResultsColumnSectionRenderer
            table_renderer:
              module_name: custom_renderers.custom_table_content_block
              class_name: CustomTableContentBlockRenderer

      profiling:  # if empty, or one of ['0', 'None', 'False', 'false', 'FALSE', 'none
↪', 'NONE'], section not rendered
```

To support that custom renderer, we need to ensure the implementation is available in our plugins/ directory. Note that we can use a subdirectory and standard python submodule notation, but that we need to include an __init__.py file in our custom_renderers package.

```
plugins/
├── custom_renderers
│   ├── __init__.py
│   └── custom_table_content_block.py
└── additional_ge_plugin.py
```

Building Data Docs

Using the CLI

The *Great Expectations CLI* can build comprehensive Data Docs from expectation suites available to the configured context and validations available in the `great_expectations/uncommitted` directory.

```
great_expectations docs build
```

When called without additional arguments, this command will render all the Data Docs sites specified in great_expectations.yml configuration file into HTML and open them in a web browser.

The command will print out the locations of index.html file for each site.

To disable the web browser opening behavior use *–no-view* option.

To render just one site, use *–site-name SITE_NAME* option.

Here is when the `docs build` command should be called:

- when you want to fully rebuild a Data Docs site
- after a new expectation suite is added or an existing one is edited
- after new data is profiled (only if you declined the prompt to build data docs when running the profiling command)

When a new validation result is generated after running a Validation Operator, the Data Docs sites will add this result automatically if the operator has the UpdateDataDocsAction action configured (read *Actions*).

Using the raw API

The underlying python API for rendering documentation is still new and evolving. Use the following snippet as a guide for how to profile a single batch of data and build documentation from the validation_result.

```
import os
import great_expectations as ge

from great_expectations.profile.basic_dataset_profiler import BasicDatasetProfiler
from great_expectations.render.renderer import ProfilingResultsPageRenderer,␣
↪ExpectationSuitePageRenderer
from great_expectations.data_context.util import safe_mmkdir
from great_expectations.render.view import DefaultJinjaPageView

profiling_html_filepath = '/path/into/which/to/save/results.html'

# obtain the DataContext object
context = ge.data_context.DataContext()

# load a batch to profile
```

(continues on next page)

(continued from previous page)

```
context.create_expectation_suite('default')
batch = context.get_batch(
  batch_kwargs=context.build_batch_kwargs("my_datasource", "my_batch_kwargs_generator
↪", "my_asset")
  expectation_suite_name='default',
)

# run the profiler on the batch - this returns an expectation suite and validation␣
↪results for this suite
expectation_suite, validation_result = BasicDatasetProfiler().profile(batch)

# use a renderer to produce a document model from the validation results
document_model = ProfilingResultsPageRenderer().render(validation_result)

# use a view to render the document model (produced by the renderer) into a HTML␣
↪document
safe_mmkdir(os.path.dirname(profiling_html_filepath))
with open(profiling_html_filepath, 'w') as writer:
    writer.write(DefaultJinjaPageView().render(document_model))
```

Customizing Data Docs

Introduction

Data Docs uses the Jinja template engine to generate HTML pages. The built-in Jinja templates used to compile Data Docs pages are implemented in the `great_expectations.render.view` module and are tied to *View* classes. Views determine how page content is displayed. The content data that Views specify and consume is generated by *Renderer* classes and are implemented in the `great_expectations.render.renderer` module. Renderers take Great Expectations objects as input and return typed dictionaries - Views take these dictionaries as input and output rendered HTML.

Built-In Views and Renderers

Out of the box, Data Docs supports two top-level Views (i.e. pages), `great_expectations.render.view.DefaultJinjaIndexPageView`, for a site index page, and `great_expectations.render.view.DefaultJinjaPageView` for all other pages. Pages are broken into sections - `great_expectations.render.view.DefaultJinjaSectionView` - which are composed of UI components - `great_expectations.render.view.DefaultJinjaComponentView`. Each of these Views references a single base Jinja template, which can incorporate any number of other templates through inheritance.

Data Docs comes with the following built-in site page Renderers:

- `great_expectations.render.renderer.SiteIndexPageRenderer` (index page)
- `great_expectations.render.renderer.ProfilingResultsPageRenderer` (Profiling Results pages)
- `great_expectations.render.renderer.ExpectationSuitePageRenderer` (Expectation Suite pages)
- `great_expectations.render.renderer.ValidationResultsPageRenderer` (Validation Results pages)

Analogous to the base View templates referenced above, these Renderers can be thought of as base Renderers for the primary Data Docs pages, and may call on many other ancillary Renderers.

It is possible to extend Renderers and Views and customize the particular class used to render any of the objects in your documentation.

Other Tools

In addition to Jinja, Data Docs draws on the following libraries to compile HTML documentation, which you can use to further customize HTML documentation:

- Bootstrap
- Font Awesome
- jQuery
- Altair

Getting Started

Making Minor Adjustments

Many of the HTML elements in the default Data Docs pages have pre-configured classes that you may use to make minor adjustments using your own custom CSS. By default, when you run `great_expectations init`, Great Expectations creates a scaffold within the `plugins` directory for customizing Data Docs. Within this scaffold is a file called `data_docs_custom_styles.css` - this CSS file contains all the pre-configured classes you may use to customize the look and feel of the default Data Docs pages. All custom CSS, applied to these pre-configured classes or any other HTML elements, must be placed in this file.

Scaffolded directory tree:

```
plugins
└── custom_data_docs
    ├── renderers
    ├── styles
    │   └── data_docs_custom_styles.css
    └── views
```

Using Custom Views and Renderers

Suppose you start a new Great Expectations project by running `great_expectations init` and compile your first Data Docs site. After looking over the local site, you decide you want to implement the following changes:

1. A completely new Expectation Suite page, requiring a new View and Renderer
2. A smaller modification to the default Validation page, swapping out a child renderer for a custom version
3. Remove Profiling Results pages from the documentation

To make these changes, you must first implement the custom View and Renderers and ensure they are available in the plugins directory specified in your project configuration (`plugins/` by default). Note that you can use a sub-directory and standard python submodule notation, but you must include an __init__.py file in your custom package. By default, when you run `great_expectations init`, Great Expectations creates placeholder directories for your custom views, renderers, and CSS stylesheets within the `plugins` directory. If you wish, you may save your custom views and renderers in an alternate location however, any CSS stylesheets must be saved to `plugins/custom_data_docs/styles`.

Scaffolded directory tree:

```
plugins
└── custom_data_docs
    ├── renderers
    ├── styles
    │   └── data_docs_custom_styles.css
    └── views
```

When you are done with your implementations, your `plugins/` directory has the following structure:

```
plugins
└── custom_data_docs
    ├── renderers
    │   ├── __init__.py
    │   ├── custom_table_renderer.py
    │   └── custom_expectation_suite_page_renderer.py
    ├── styles
    │   └── data_docs_custom_styles.css
    └── views
        ├── __init__.py
        └── custom_expectation_suite_view.py
```

For Data Docs to use your custom Views and Renderers when compiling your local Data Docs site, you must specify where to use them in the `data_docs_sites` section of your project configuration.

Before modifying your project configuration, the relevant section looks like this:

```
data_docs_sites:
  local_site:
    class_name: SiteBuilder
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: uncommitted/data_docs/local_site/
    site_index_builder:
      class_name: DefaultSiteIndexBuilder
```

This is what it looks like after your changes are added:

```
data_docs_sites:
  local_site:
    class_name: SiteBuilder
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: uncommitted/data_docs/local_site/
    site_index_builder:
      class_name: DefaultSiteIndexBuilder
    site_section_builders:
      expectations:
        renderer:
          module_name: custom_data_docs.renderers.custom_expectation_suite_page_
↪renderer
          class_name: CustomExpectationSuitePageRenderer
        view:
          module_name: custom_data_docs.views.custom_expectation_suite_view
          class_name: CustomExpectationSuiteView
      validations:
        renderer:
          module_name: great_expectations.render.renderer
          class_name: ValidationResultsPageRenderer
```

(continues on next page)

(continued from previous page)

```
          column_section_renderer:
            class_name: ValidationResultsColumnSectionRenderer
            table_renderer:
              module_name: custom_data_docs.renderers.custom_table_renderer
              class_name: CustomTableRenderer
      profiling:
```

By providing an empty `profiling` key within `site_section_builders`, your third goal is achieved and Data Docs will no longer render Profiling Results pages. The same can be achieved by setting the `profiling` key to any of the following values: `['0', 'None', 'False', 'false', 'FALSE', 'none', 'NONE']`.

Lastly, to compile your newly-customized Data Docs local site, you run `great_expectations docs build` from the command line.

`site_section_builders` defaults:

```
site_section_builders:
  expectations: # if empty, or one of ['0', 'None', 'False', 'false', 'FALSE', 'none',
↪ 'NONE'], section not rendered
    class_name: DefaultSiteSectionBuilder
    source_store_name: expectations_store
    renderer:
      module_name: great_expectations.render.renderer
      class_name: ExpectationSuitePageRenderer

  validations: # if empty, or one of ['0', 'None', 'False', 'false', 'FALSE', 'none',
↪'NONE'], section not rendered
    class_name: DefaultSiteSectionBuilder
    source_store_name: validations_store
    run_id_filter:
      ne: profiling # exclude validations with run id "profiling" - reserved for␣
↪profiling results
    renderer:
      module_name: great_expectations.render.renderer
      class_name: ValidationResultsPageRenderer

  profiling: # if empty, or one of ['0', 'None', 'False', 'false', 'FALSE', 'none',
↪'NONE'], section not rendered
    class_name: DefaultSiteSectionBuilder
    source_store_name: validations_store
    run_id_filter:
      eq: profiling
    renderer:
      module_name: great_expectations.render.renderer
      class_name: ProfilingResultsPageRenderer
```

Re-Purposing Built-In Views

If you would like to re-purpose a built-in View, you may do so by implementing a custom renderer that outputs an appropriately typed and structured dictionary for that View.

Built-in Views and corresponding input types:

- `great_expectations.render.view.DefaultJinjaPageView`: `great_expectations.render.types.RenderedDocumentContent`
- `great_expectations.render.view.DefaultJinjaSectionView`: `great_expectations.render.types.RenderedSectionContent`
- `great_expectations.render.view.DefaultJinjaComponentView`: `great_expectations.render.types.RenderedComponentContent`

An example of a custom page Renderer, using all built-in UI elements is provided below.

Custom Page Renderer Example

```
import altair as alt
import pandas as pd

from great_expectations.render.renderer.renderer import Renderer
from great_expectations.render.types import (
    RenderedDocumentContent,
    RenderedSectionContent,
    RenderedComponentContent,
    RenderedHeaderContent, RenderedBulletListContent, RenderedTableContent,␣
↪RenderedStringTemplateContent,
    RenderedGraphContent, ValueListContent)


class CustomPageRenderer(Renderer):
    @classmethod
    def _get_header_content_block(cls, header="", subheader="", highlight=True):
        return RenderedHeaderContent(**{
            "content_block_type": "header",
            "header": RenderedStringTemplateContent(**{
                "content_block_type": "string_template",
                "string_template": {
                    "template": header,
                }
            }),
            "subheader": subheader,
            "styling": {
                "classes": ["col-12"],
                "header": {
                    "classes": ["alert", "alert-secondary"] if highlight else []
                }
            }
        })

    @classmethod
    def _get_bullet_list_content_block(cls, header="", subheader="", col=12):
        return RenderedBulletListContent(**{
            "content_block_type": "bullet_list",
```

(continues on next page)

(continued from previous page)

```
            "header": header,
            "subheader": subheader,
            "bullet_list": [
                "Aenean porttitor turpis.",
                "Curabitur ligula urna.",
                cls._get_header_content_block(header="nested header content block",␣
↪subheader="subheader",
                                              highlight=False)
            ],
            "styling": {
                "classes": ["col-{}".format(col)],
                "styles": {
                    "margin-top": "20px"
                },
            },
        })

    @classmethod
    def _get_table_content_block(cls, header="", subheader="", col=12):
        return RenderedTableContent(**{
            "content_block_type": "table",
            "header": header,
            "subheader": subheader,
            "table": [
                ["", "column_1", "column_2"],
                [
                    "row_1",
                    cls._get_bullet_list_content_block(subheader="Nested Bullet List␣
↪Content Block"),
                    "buffalo"
                ],
                ["row_2", "crayon", "derby"],
            ],
            "styling": {
                "classes": ["col-{}".format(col), "table-responsive"],
                "styles": {
                    "margin-top": "20px"
                },
                "body": {
                    "classes": ["table", "table-sm"]
                }
            },
        })

    @classmethod
    def _get_graph_content_block(cls, header="", subheader="", col=12):
        df = pd.DataFrame({
            "value": [1, 2, 3, 4, 5, 6],
            "count": [123, 232, 543, 234, 332, 888]
        })
        bars = alt.Chart(df).mark_bar(size=20).encode(
            y='count:Q',
            x="value:O"
        ).properties(height=200, width=200, autosize="fit")
        chart = bars.to_json()

        return RenderedGraphContent(**{
```

(continues on next page)

(continued from previous page)

```
            "content_block_type": "graph",
            "header": header,
            "subheader": subheader,
            "graph": chart,
            "styling": {
                "classes": ["col-{}".format(col)],
                "styles": {
                    "margin-top": "20px"
                },
            },
        })

    @classmethod
    def _get_tooltip_string_template_content_block(cls):
        return RenderedStringTemplateContent(**{
            "content_block_type": "string_template",
            "string_template": {
                "template": "This is a string template with tooltip, using a top-
↪level custom tag.",
                "tag": "code",
                "tooltip": {
                    "content": "This is the tooltip content."
                }
            },
            "styling": {
                "classes": ["col-12"],
                "styles": {
                    "margin-top": "20px"
                },
            },
        })

    @classmethod
    def _get_string_template_content_block(cls):
        return RenderedStringTemplateContent(**{
            "content_block_type": "string_template",
            "string_template": {
                "template": "$icon This is a Font Awesome Icon, using a param-level␣
↪custom tag\n$red_text\n$bold_serif",
                "params": {
                    "icon": "",
                    "red_text": "And this is red text!",
                    "bold_serif": "And this is big, bold serif text using style␣
↪attribute..."
                },
                "styling": {
                    "params": {
                        "icon": {
                            "classes": ["fas", "fa-check-circle", "text-success"],
                            "tag": "i"
                        },
                        "red_text": {
                            "classes": ["text-danger"]
                        },
                        "bold_serif": {
                            "styles": {
                                "font-size": "22px",
```

(continues on next page)

(continued from previous page)

```
                                "font-weight": "bold",
                                "font-family": "serif"
                            }
                        }
                    }
                }
            },
            "styling": {
                "classes": ["col-12"],
                "styles": {
                    "margin-top": "20px"
                },
            },
        })

    @classmethod
    def _get_value_list_content_block(cls, header="", subheader="", col=12):
        return ValueListContent(**{
            "content_block_type": "value_list",
            "header": header,
            "subheader": subheader,
            "value_list": [{
                "content_block_type": "string_template",
                "string_template": {
                    "template": "$value",
                    "params": {
                        "value": value
                    },
                    "styling": {
                        "default": {
                            "classes": ["badge", "badge-info"],
                        },
                    }
                }
            } for value in [
                "Andrew",
                "Elijah",
                "Matthew",
                "Cindy",
                "Pam"
            ]],
            "styling": {
                "classes": ["col-{}".format(col)]
            },
        })

    @classmethod
    def render(cls, ge_dict={}):
        return RenderedDocumentContent(**{
            "renderer_type": "CustomValidationResultsPageRenderer",
            "data_asset_name": "my_data_asset_name",
            "full_data_asset_identifier": "my_datasource/my_generator/my_generator_
↪asset",
            "page_title": "My Page Title",
            "sections": [
                RenderedSectionContent(**{
                    "section_name": "Header Content Block",
```

(continues on next page)

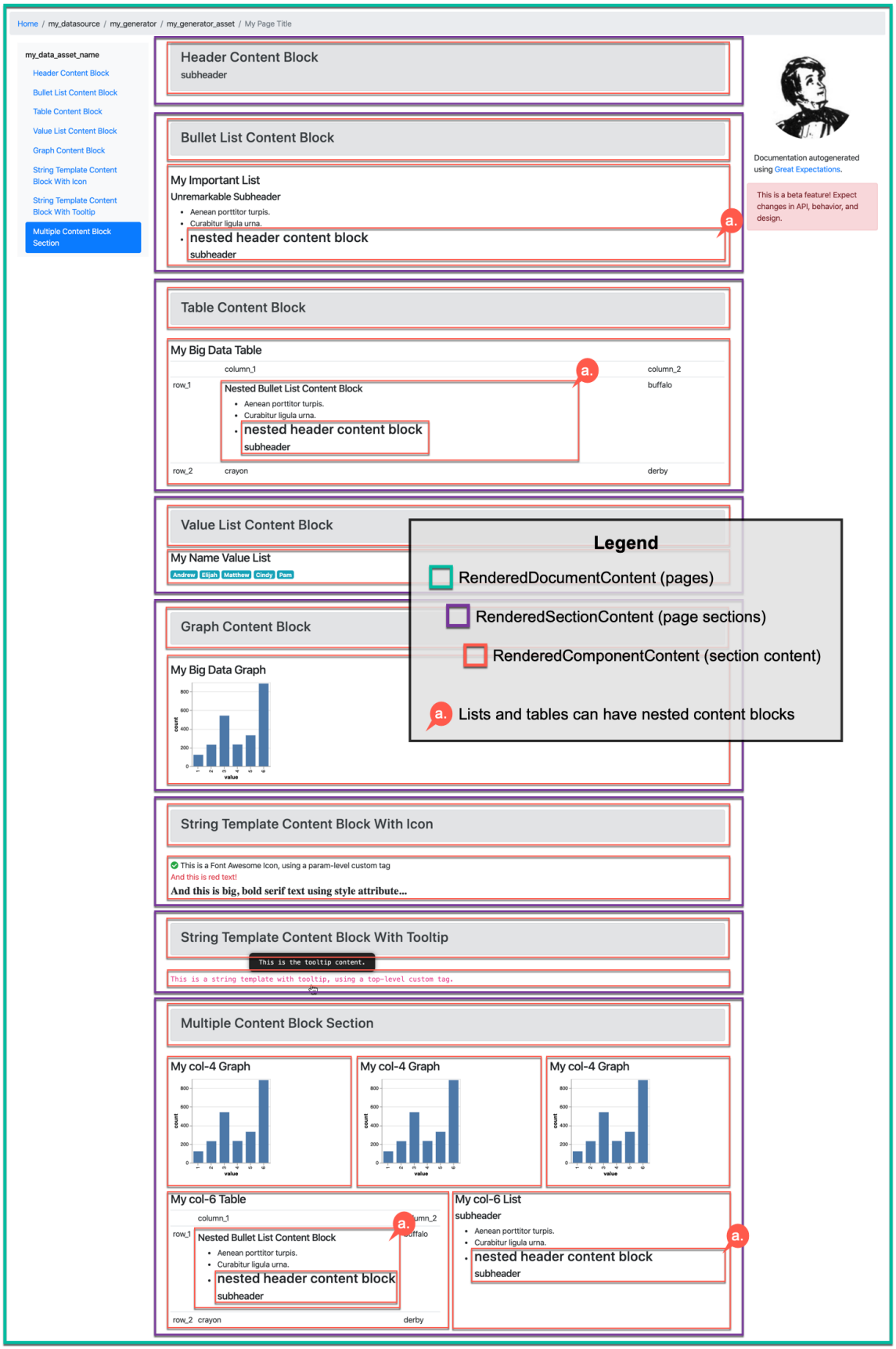
(continued from previous page)

```
                    "content_blocks": [
                        cls._get_header_content_block(header="Header Content Block",␣
↪subheader="subheader")]
                }),
                RenderedSectionContent(**{
                    "section_name": "Bullet List Content Block",
                    "content_blocks": [
                        cls._get_header_content_block(header="Bullet List Content␣
↪Block"),
                        cls._get_bullet_list_content_block(header="My Important List",
                                                           subheader="Unremarkable␣
↪Subheader")
                    ]
                }),
                RenderedSectionContent(**{
                    "section_name": "Table Content Block",
                    "content_blocks": [
                        cls._get_header_content_block(header="Table Content Block"),
                        cls._get_table_content_block(header="My Big Data Table"),
                    ]
                }),
                RenderedSectionContent(**{
                    "section_name": "Value List Content Block",
                    "content_blocks": [
                        cls._get_header_content_block(header="Value List Content Block
↪"),
                        cls._get_value_list_content_block(header="My Name Value List
↪"),
                    ]
                }),
                RenderedSectionContent(**{
                    "section_name": "Graph Content Block",
                    "content_blocks": [
                        cls._get_header_content_block(header="Graph Content Block"),
                        cls._get_graph_content_block(header="My Big Data Graph"),
                    ]
                }),
                RenderedSectionContent(**{
                    "section_name": "String Template Content Block With Icon",
                    "content_blocks": [
                        cls._get_header_content_block(header="String Template Content␣
↪Block With Icon"),
                        cls._get_string_template_content_block()
                    ]
                }),
                RenderedSectionContent(**{
                    "section_name": "String Template Content Block With Tooltip",
                    "content_blocks": [
                        cls._get_header_content_block(header="String Template Content␣
↪Block With Tooltip"),
                        cls._get_tooltip_string_template_content_block()
                    ]
                }),
                RenderedSectionContent(**{
                    "section_name": "Multiple Content Block Section",
                    "content_blocks": [
                        cls._get_header_content_block(header="Multiple Content Block␣
↪Section"),
```

(continues on next page)

(continued from previous page)

```
                        cls._get_graph_content_block(header="My col-4 Graph", col=4),
                        cls._get_graph_content_block(header="My col-4 Graph", col=4),
                        cls._get_graph_content_block(header="My col-4 Graph", col=4),
                        cls._get_table_content_block(header="My col-6 Table", col=6),
                        cls._get_bullet_list_content_block(header="My col-6 List",␣
↪subheader="subheader", col=6)
                    ]
                }),
            ]
        })
```


Home / my_datasource / my_generator / my_generator_asset / My Page Title
my_data_asset_name
Header Content Block
Bullet List Content Block
Table Content Block
Value List Content Block
Graph Content Block
String Template Content Block With Icon
String Template Content Block With Tooltip
Multiple Content Block Section
Header Content Block
subheader
Bullet List Content Block
My Important List
Unremarkable Subheader
Aenean porttitor turpis.
Curabitur ligula urna.
nested header content block
subheader
a.
Table Content Block
My Big Data Table
column_1
column_2
row_1
buffalo
Nested Bullet List Content Block
Aenean porttitor turpis.
Curabitur ligula urna.
nested header content block
subheader
row_2
crayon
derby
Value List Content Block
My Name Value List
Andrew
Elijah
Matthew
Cindy
Pam
Graph Content Block
My Big Data Graph
count
value
Documentation autogenerated using Great Expectations.
This is a beta feature! Expect changes in API, behavior, and design.
Legend
RenderedDocumentContent (pages)
RenderedSectionContent (page sections)
RenderedComponentContent (section content)
Lists and tables can have nested content blocks
String Template Content Block With Icon
This is a Font Awesome Icon, using a param-level custom tag
And this is red text!
And this is big, bold serif text using style attribute...
String Template Content Block With Tooltip
This is the tooltip content.
This is a string template with tooltip, using a top-level custom tag.
Multiple Content Block Section
My col-4 Graph
My col-4 Graph
My col-4 Graph
My col-6 Table
My col-6 List
subheader

Dependencies

- Font Awesome 5.10.1
- Bootstrap 4.3.1
- jQuery 3.2.1
- altair 3.1.0
- Vega 5.3.5
- Vega-Lite 3.2.1
- Vega-Embed 4.0.0

Data Docs is implemented in the `great_expectations.render` module.

7.3.3 Profiling Reference

Profiling produces a special kind of *Data Docs* that are purely descriptive.

Expectations and Profiling

In order to characterize a data asset, Profiling uses an Expectation Suite. Unlike the Expectations that are typically used for data validation, these expectations do not necessarily apply any constraints; they can simply identify statistics or other data characteristics that should be evaluated and made available in GE. For example, when the `BasicDatasetProfiler` encounters a numeric column, it will add an `expect_column_mean_to_be_between` expectation but choose the min_value and max_value to both be None: essentially only saying that it expects a mean to exist.

```
{
  "expectation_type": "expect_column_mean_to_be_between",
  "kwargs": {
    "column": "rating",
    "min_value": null,
    "max_value": null
  }
}
```

To "profile" a datasource, therefore, the *`BasicDatasetProfiler`* included in GE will generate a large number of very loosely-specified expectations. Effectively it is asserting that the given statistic is relevant for evaluating batches of that data asset, but it is not yet sure what the statistic's value should be.

In addition to creating an expectation suite, profiling data tests the suite against data. The validation_result contains the output of that expectation suite when validated against the same batch of data. For a loosely specified expectation like in our example above, getting the observed value was the sole purpose of the expectation.

```
{
  "success": true,
  "result": {
    "observed_value": 4.05,
    "element_count": 10000,
    "missing_count": 0,
    "missing_percent": 0
  }
}
```

Running a profiler on a data asset can also be useful to produce a large number of expectations to review and potentially transfer to a new expectation suite used for validation in a pipeline.

How to Run Profiling

Run During Init

The `great_expectations init` command will auto-generate an example Expectation Suite using a very basic profiler that quickly glances at 1,000 rows of your data. This is not a production suite - it is only meant to show examples of Expectations, many of which may not be meaningful.

Expectation Suites generated by the profiler will be saved in the configured `expectations` directory for Expectation Suites. The Expectation Suite name by default is the name of the profiler that generated it. Validation results will be saved in the `uncommitted/validations` directory by default. When profiling is complete, Great Expectations will build and launch Data Docs based on your data.

Run From Command Line

The GE command-line interface can profile a datasource:

```
great_expectations datasource profile DATASOURCE_NAME
```

Expectation Suites generated by the profiler will be saved in the configured `expectations` directory for Expectation Suites. The Expectation Suite name by default is the name of the profiler that generated it. Validation results will be saved in the `uncommitted/validations` directory by default. When profiling is complete, Great Expectations will build and launch Data Docs based on your data.

See *Data Docs* for more information.

Run From Jupyter Notebook

If you want to profile just one data asset in a datasource (e.g., one table in the database), you can do it using Python in a Jupyter notebook:

```
from great_expectations.profile.basic_dataset_profiler import BasicDatasetProfiler

# obtain the DataContext object
context = ge.data_context.DataContext()

# load a batch from the data asset
batch = context.get_batch('ratings')

# run the profiler on the batch - this returns an expectation suite and validation␣
↪results for this suite
expectation_suite, validation_result = BasicDatasetProfiler.profile(batch)

# save the resulting expectation suite with a custom name
context.save_expectation_suite(expectation_suite, "ratings", "my_profiled_expectations
↪")
```

Custom Profilers

Like most things in Great Expectations, Profilers are designed to be extensibile. You can develop your own profiler by subclassing `DatasetProfiler`, or from the parent `DataAssetProfiler` class itself. For help, advice, and ideas on developing custom profilers, please get in touch on the Great Expectations slack channel.

Profiling Limitations

Inferring Data Types

When profiling CSV files, the profiler makes assumptions, such as considering the first line to be the header. Overriding these assumptions is currently possible only when running profiling in Python by passing extra arguments to get_batch.

Data Samples

Since profiling and expectations are so tightly linked, getting samples of *expected* data requires a slightly different approach than the normal path for profiling. Stay tuned for more in this area!

7.3.4 Migrating Between Versions

While we are committed to keeping Great Expectations as stable as possible, sometimes breaking changes are necessary to maintain our trajectory. This is especially true as the library has evolved from just a data quality tool to a more capable framework including data docs and profiling in addition to validation.

Great Expectations provides a warning when the currently-installed version is different from the version stored in the expectation suite.

Since expectation semantics are usually consistent across versions, there is little change required when upgrading great expectations, with some exceptions noted here.

Using the project check-config Command

To facilitate this substantial config format change, starting with version 0.8.0 we introduced `project check-config` to sanity check your config files. From your project directory, run:

```
great_expectations project check-config
```

This can be used at any time and will grow more robust and helpful as our internal config typing system improves.

You will most likely be prompted to install a new template. Rest assured that your original yaml file will be archived automatically for you. Even so, it's in your source control system already, right? ;-)

Upgrading to 0.9.x

In the 0.9.0 release, there are several changes to the DataContext API.

Follow these steps to upgrade your existing Great Expectations project:

- In the terminal navigate to the parent of the `great_expectations` directory of your project.
- Run this command:

```
great_expectations project check-config
```

- For every item that needs to be renamed the command will display a message that looks like this: `The class name 'X' has changed to 'Y'`. Replace all occurrences of X with Y in your project's `great_expectations.yml` config file.
- After saving the config file, rerun the check-config command.
- Depending on your configuration, you will see 3-6 of these messages.
- The command will display this message when done: `Your config file appears valid!`.
- Rename your Expectation Suites to make them compatible with the new naming. Save this Python code snippet in a file called `update_project.py`, then run it using the command: `python update_project.py PATH_TO_GE_CONFIG_DIRECTORY`:

```
#!/usr/bin/env python3
import sys
import os
import json
import uuid
import shutil
def update_validation_result_name(validation_result):
    data_asset_name = validation_result["meta"].get("data_asset_name")
    if data_asset_name is None:
        print("    No data_asset_name in this validation result. Unable to update it.
↪")
        return
    data_asset_name_parts = data_asset_name.split("/")
    if len(data_asset_name_parts) != 3:
        print("    data_asset_name in this validation result does not appear to be␣
↪normalized. Unable to update it.")
        return
    expectation_suite_suffix = validation_result["meta"].get("expectation_suite_name")
    if expectation_suite_suffix is None:
        print("    No expectation_suite_name found in this validation result. Unable␣
↪to update it.")
        return
    expectation_suite_name = ".".join(
        data_asset_name_parts +
        [expectation_suite_suffix]
    )
    validation_result["meta"]["expectation_suite_name"] = expectation_suite_name
    try:
        del validation_result["meta"]["data_asset_name"]
    except KeyError:
        pass
def update_expectation_suite_name(expectation_suite):
    data_asset_name = expectation_suite.get("data_asset_name")
    if data_asset_name is None:
```

(continues on next page)

(continued from previous page)

```
        print("    No data_asset_name in this expectation suite. Unable to update it.
↪")
        return
    data_asset_name_parts = data_asset_name.split("/")
    if len(data_asset_name_parts) != 3:
        print("    data_asset_name in this expectation suite does not appear to be␣
↪normalized. Unable to update it.")
        return
    expectation_suite_suffix = expectation_suite.get("expectation_suite_name")
    if expectation_suite_suffix is None:
        print("    No expectation_suite_name found in this expectation suite. Unable␣
↪to update it.")
        return
    expectation_suite_name = ".".join(
        data_asset_name_parts +
        [expectation_suite_suffix]
    )
    expectation_suite["expectation_suite_name"] = expectation_suite_name
    try:
        del expectation_suite["data_asset_name"]
    except KeyError:
        pass
def update_context_dir(context_root_dir):
    # Update expectation suite names in expectation suites
    expectations_dir = os.path.join(context_root_dir, "expectations")
    for subdir, dirs, files in os.walk(expectations_dir):
        for file in files:
            if file.endswith(".json"):
                print("Migrating suite located at: " + str(os.path.join(subdir,␣
↪file)))
                with open(os.path.join(subdir, file), 'r') as suite_fp:
                    suite = json.load(suite_fp)
                update_expectation_suite_name(suite)
                with open(os.path.join(subdir, file), 'w') as suite_fp:
                    json.dump(suite, suite_fp)
    # Update expectation suite names in validation results
    validations_dir = os.path.join(context_root_dir, "uncommitted", "validations")
    for subdir, dirs, files in os.walk(validations_dir):
        for file in files:
            if file.endswith(".json"):
                print("Migrating validation_result located at: " + str(os.path.
↪join(subdir, file)))
                try:
                    with open(os.path.join(subdir, file), 'r') as suite_fp:
                        suite = json.load(suite_fp)
                    update_validation_result_name(suite)
                    with open(os.path.join(subdir, file), 'w') as suite_fp:
                        json.dump(suite, suite_fp)
                    try:
                        run_id = suite["meta"].get("run_id")
                        es_name = suite["meta"].get("expectation_suite_name").split(".
↪")

                        filename = "converted__" + str(uuid.uuid1()) + ".json"
                        os.makedirs(os.path.join(
                            context_root_dir, "uncommitted", "validations",
                            *es_name, run_id
                        ), exist_ok=True)
```

(continues on next page)

(continued from previous page)

```
                        shutil.move(os.path.join(subdir, file),
                                    os.path.join(
                                        context_root_dir, "uncommitted", "validations
↪",
                                        *es_name, run_id, filename
                                    )
                        )
                    except OSError as e:
                        print("    Unable to move validation result; file has been␣
↪updated to new "
                              "format but not moved to new store location.")
                    except KeyError:
                        pass  # error will have been generated above
                except json.decoder.JSONDecodeError:
                    print("    Unable to process file: error reading JSON.")
if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Please provide a path to update.")
        sys.exit(-1)
    path = str(os.path.abspath(sys.argv[1]))
    print("About to update context dir for path: " + path)
    update_context_dir(path)
```

- Rebuild Data Docs:

```
great_expectations docs build
```

- This project has now been migrated to 0.9.0. Please see the list of changes below for more detailed information.

CONFIGURATION CHANGES:

- FixedLengthTupleXXXX stores are renamed to TupleXXXX stores; they no longer allow or require (or allow) a key_length to be specified, but they do allow *filepath_prefix* and/or *filepath_suffix* to be configured as an alternative to an the *filepath_template*.
- ExtractAndStoreEvaluationParamsAction is renamed to StoreEvaluationParametersAction; a new StoreMetricsAction is available as well to allow DataContext-configured metrics to be saved.
- The InMemoryEvaluationParameterStore is replaced with the EvaluationParameterStore; EvaluationParameterStore and MetricsStore can both be configured to use DatabaseStoreBackend instead of the InMemoryStoreBackend.
- The *type* key can no longer be used in place of class_name in configuration. Use *class_name* instead.
- BatchKwargsGenerators are more explicitly named; we avoid use of the term "Generator" because it is ambiguous. All existing BatchKwargsGenerators have been renamed by substituting "BatchKwargsGenerator" for "Generator"; for example GlobReaderGenerator is now GlobReaderBatchKwargsGenerator.
- ReaderMethod is no longer an enum; it is a string of the actual method to be invoked (e.g. *read_csv* for pandas). That change makes it easy to specify arbitrary reader_methods via batch_kwargs (including read_pickle), BUT existing configurations using enum-based reader_method in batch_kwargs will need to update their code. For example, a pandas datasource would use *reader_method: read_csv`* instead of *reader_method: csv*

CODE CHANGES:

- DataAssetName and name normalization have been completely eliminated, which causes several related changes to code using the DataContext.
  - data_asset_name is **no longer** a parameter in the create_expectation_suite, get_expectation_suite, or get_batch commands; expectation suite names exist in an independent namespace.

  - batch_kwargs alone now define the batch to be received, and the datasource name **must** be included in batch_kwargs as the "datasource" key.
  - **A generator name is therefore no longer required to get data or define an expectation suite.**
  - The BatchKwargsGenerators API has been simplified; *build_batch_kwargs* should be the entrypoint for all cases of using a generator to get batch_kwargs, including when explicitly specifying a partition, limiting the number of returned rows, accessing saved kwargs, or using any other BatchKwargsGenerator feature. BatchKwargsGenerators *must* be attached to a specific datasource to be instantiated.
  - This tutorial uses the latest API for validating data: *Validate Data*

- **Database store tables are not compatible** between versions and require a manual migration; the new default table names are: *ge_validations_store*, *ge_expectations_store*, *ge_metrics*, and *ge_evaluation_parameters*. The Validations Store uses a three-part compound primary key consisting of run_id, expectation_suite_name, and batch_identifier; Expectations Store uses the expectation_suite_name as its only key. Both Metrics and Evaluation Parameters stores use *run_id*, *expectation_suite_name*, *metric_id*, and *metric_kwargs_id* to form a compound primary key.
- The term "batch_fingerprint" is no longer used, and has been replaced with "batch_markers". It is a dictionary that, like batch_kwargs, can be used to construct an ID.
- *get_data_asset_name* and *save_data_asset_name* are removed.
- There are numerous under-the-scenes changes to the internal types used in GreatExpectations. These should be transparent to users.

Upgrading to 0.8.x

In the 0.8.0 release, our DataContext config format has changed dramatically to enable new features including extensibility.

Some specific changes:

- New top-level keys:
  - *expectations_store_name*
  - *evaluation_parameter_store_name*
  - *validations_store_name*
- Deprecation of the *type* key for configuring objects (replaced by *class_name* (and *module_name* as well when ambiguous).
- Completely new *SiteBuilder* configuration. See *Data Docs Reference*.

BREAKING:

  - **top-level `validate` has a new signature**, that offers a variety of different options for specifying the DataAsset class to use during validation, including *data_asset_class_name* / *data_asset_module_name* or *data_asset_class*
  - Internal class name changes between alpha versions: - InMemoryEvaluationParameterStore - ValidationsStore - ExpectationsStore - ActionListValidationOperator
  - Several modules are now refactored into different names including all datasources
  - InMemoryBatchKwargs use the key dataset instead of df to be more explicit

Pre-0.8.x configuration files `great_expectations.yml` are not compatible with 0.8.x. Run `great_expectations project check-config` - it will offer to create a new config file. The new config file will not have any customizations you made, so you will have to copy these from the old file.

If you run into any issues, please ask for help on Slack.

Upgrading to 0.7.x

In version 0.7, GE introduced several new features, and significantly changed the way DataContext objects work:

- A *DataContexts* object manages access to expectation suites and other configuration in addition to data assets. It provides a flexible but opinionated structure for creating and storing configuration and expectations in version control.
- When upgrading from prior versions, the new *Datasources* objects provide the same functionality that compute-environment-specific data context objects provided before, but with significantly more flexibility.
- The term "autoinspect" is no longer used directly, having been replaced by a much more flexible *Profiling* feature.

7.4 Integrations

7.4.1 Redshift

To add a Redshift datasource do this:

1. Run `great_expectations datasource new`
2. Choose the *SQL* option from the menu.
3. When asked which sqlalchemy driver to use enter `redshift`.
4. Consult the SQLAlchemy docs for help building a connection string for your Redshift cluster. It will look something like this:

   Note we have had better luck so far using postgres dialect instead of redshift dialect.

   ```
   "postgresql+psycopg2://username:password@my_redshift_endpoint.us-east-2.
   ↪redshift.amazonaws.com:5439/my_database?sslmode=require"
   ```

5. Paste in this connection string and finish out the cli prompts.
6. Should you need to modify your connection string you can manually edit the `great_expectations/uncommitted/config_variables.yml` file.

Additional Notes

Depending on your Redshift cluster configuration, you may or may not need the `sslmode` parameter at the end of the connection url. You can delete everything after the `?` in the connection string above.

7.4.2 BigQuery

To add a BigQuery datasource do this:

1. Run `great_expectations datasource new`
2. Choose the *SQL* option from the menu.
3. When asked which sqlalchemy driver to use enter `bigquery`.
4. Consult the **`PyBigQuery <https://github.com/mxmzdlv/pybigquery`_** docs for help building a connection string for your BigQuery cluster. It will look something like this:

```
"bigquery://project-name"
```

5. Paste in this connection string and finish out the interactive prompts.
6. Should you need to modify your connection string you can manually edit the `great_expectations/uncommitted/config_variables.yml` file.

Custom Queries with SQL datasource

While other backends use temporary tables to generate batches of data from custom queries, BigQuery does not support ephemeral temporary tables. As a work-around, GE will create or replace a *permanent table* when the user supplies a custom query.

Users can specify a table via a Batch Kwarg called `bigquery_temp_table`:

```
batch_kwargs = {
    "query": "SELECT * FROM `my-project.my_dataset.my_table`",
    "bigquery_temp_table": "my_other_dataset.temp_table"
}
```

Otherwise, default behavior depends on how the pybigquery engine is configured:

If a default BigQuery dataset is defined in the connection string (for example, `bigquery://project-name/dataset-name`), and no `bigquery_temp_table` Batch Kwarg is supplied, then GE will create a permanent table with a random UUID in that location (e.g. `project-name.dataset-name.ge_tmp_1a1b6511_03e6_4e18_a1b2_d85f9e9045c3`).

If a default BigQuery dataset is not defined in the connection string (for example, `bigquery://project-name`) and no `bigquery_temp_table` Batch Kwawrg is supplied, then custom queries will fail.

Additional Notes

Follow the Google Cloud library guide for authentication.

Install the pybigquery package for the BigQuery sqlalchemy dialect (`pip install pybigquery`)

7.4.3 Snowflake

When using the snowflake dialect, *SqlAlchemyDataset* will create a **transient** table instead of a **temporary** table when passing in *query* Batch Kwargs or providing *custom_sql* to its constructor. Consequently, users **must** provide a *snowflake_transient_table* in addition to the *query* parameter. Any existing table with that name will be overwritten.

7.5 Advanced Features

7.5.1 Evaluation Parameters

Often, the specific parameters associated with an expectation will be derived from upstream steps in a processing pipeline. For example, we may want to *expect_table_row_count_to_equal* a value stored in a previous step.

Great Expectations makes it possible to use "Evaluation Parameters" to accomplish that goal. We declare Expectations using parameters that need to be provided at validation time. During interactive development, we can even provide a temporary value that should be used during the initial evaluation of the expectation.

```
>>> my_df.expect_table_row_count_to_equal(
...     value={"$PARAMETER": "upstream_row_count", "$PARAMETER.upstream_row_count": 10}
↪,
...     result_format={'result_format': 'BOOLEAN_ONLY'})
{
  'success': True
}
```

More typically, when validating expectations, you can provide evaluation parameters that are only available at runtime:

```
my_df.validate(expectation_suite=my_dag_step_config, evaluation_parameters={"upstream_
↪row_count": upstream_row_count})
```

Evaluation Parameter Expressions

In many cases, evaluation parameters are most useful when they can allow a range of values. For example, we might want to specify that a new table's row count should be between 90 - 110 % of an upstream table's row count (or a count from a previous run). Evaluation parameters support basic arithmetic expressions to accomplish that goal:

```
>>> my_df.expect_table_row_count_to_be_between(
...     min_value={"$PARAMETER": "trunc(upstream_row_count * 0.9)"},
...     max_value={"$PARAMETER": "trunc(upstream_row_count * 1.1)"},
...     result_format={'result_format': 'BOOLEAN_ONLY'})
{
  'success': True
}
```

Evaluation parameters are not limited to simple values, for example you could include a list as a parameter value:

```
my_df.column_values_to_be_in_set("my_column", value_set={"$PARAMETER": "runtime_values
↪"})
my_df.validate(evaluation_parameters={"runtime_values": [1, 2, 3])
```

However, it is not possible to mix complex values with arithmetic expressions.

Storing Evaluation Parameters

Data Context Evaluation Parameter Store

A DataContext can automatically identify and store evaluation parameters that are referenced in other expectation suites. The evaluation parameter store uses a URN schema for identifying dependencies between expectation suites.

The DataContext-recognized URN must begin with the string `urn:great_expectations`. Valid URNs must have one of the following structures to be recognized by the Great Expectations DataContext:

```
urn:great_expectations:validations:<expectation_suite_name>:<metric_name>
urn:great_expectations:validations:<expectation_suite_name>:<metric_name>:<metric_
↪kwargs_id>
```

Replace names in <> with the desired name. For example:

```
urn:great_expectations:validations:dickens_data:expect_column_proportion_of_unique_
↪values_to_be_between.result.observed_value:column=Title
```

Storing Parameters in an Expectation Suite

You can also store parameter values in a special dictionary called evaluation_parameters that is stored in the expectation_suite to be available to multiple expectations or while declaring additional expectations.

```
>>> my_df.set_evaluation_parameter("upstream_row_count", 10)
>>> my_df.get_evaluation_parameter("upstream_row_count")
```

If a parameter has been stored, then it does not need to be provided for a new expectation to be declared:

```
>>> my_df.set_evaluation_parameter("upstream_row_count", 10)
>>> my_df.expect_table_row_count_to_be_between(max_value={"$PARAMETER": "upstream_row_
↪count"})
```

7.5.2 Data Asset Features Guide

This document describes useful features of the DataAsset object. A *DataAsset* in Great Expectations is the root class that enables declaring and validating expectations; it brings together data and expectation evaluation logic.

Interactive Evaluation

Setting the *interactive_evaluation* flag on a DataAsset make it possible to declare expectations and store expectations without immediately evaluating them. When interactive evaluation is disabled, the running an expectation method on a DataAsset will return the configuration just added to its expectation suite rather than a result object.

At initialization

```
import great_expectations as ge
import pandas as pd
df = pd.read_csv("../tests/examples/titanic.csv")
ge_df = ge.dataset.PandasDataset(df, interactive_evaluation=False)
ge_df.expect_column_values_to_be_in_set('Sex', ["male", "female"])

{
    'stored_configuration': {
        'expectation_type': 'expect_column_values_to_be_in_set',
        'kwargs': {
            'column': 'Sex',
            'value_set': ['male', 'female'],
            'result_format': 'BASIC'
        }
    }
}
```

Dynamically adjusting interactive evaluation

```
>> import great_expectations as ge
>> import pandas as pd
>> df = pd.read_csv("./tests/examples/titanic.csv")
>> ge_df = ge.dataset.PandasDataset(df, interactive_evaluation=True)
>> ge_df.expect_column_values_to_be_in_set('Sex', ["male", "female"])

{
    'success': True,
    'result': {
        'element_count': 1313,
        'missing_count': 0,
        'missing_percent': 0.0,
        'unexpected_count': 0,
        'unexpected_percent': 0.0,
        'unexpected_percent_nonmissing': 0.0,
        'partial_unexpected_list': []
    }
}

>> ge_df.set_config_value("interactive_evaluation", False)
>> ge_df.expect_column_values_to_be_in_set("PClass", ["1st", "2nd", "3rd"])

{
  'stored_configuration': {
    'expectation_type': 'expect_column_values_to_be_in_set',
    'kwargs': {
      'column': 'PClass',
      'value_set': [
        '1st',
        '2nd',
        '3rd'
      ],
      'result_format': 'BASIC'
    }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

7.5.3 Stores

Stores require a *DataContexts* which manages their creation and configuration. A store provides an abstraction for getting and setting key values in the GE ecosystem.

7.5.4 Batch Identifiers

Batch identifiers make it possible to integrate easily with other data versioning, provenance, and lineage metadata stores.

When data assets are fetched from a `Datasource`, they have *Batch Kwargs*, *Batch Id*, and *Batch Fingerprint* properties which an help track those properties.

Batch Kwargs

See the *Batch Kwargs* for more information.

Batch Id

Batch Fingerprint

7.5.5 Validation Operators

ActionListValidationOperator

ActionListValidationOperator validates each batch in its *run* method's *assets_to_validate* argument against the expectation suite included within that batch.

Then it invokes a list of configured actions on every validation result.

Each action in the list must be an instance of ValidationAction class (or its descendants). Read more about actions here: *Actions*.

The init command includes this operator in the default configuration file.

Configuration

An instance of ActionListValidationOperator is included in the default configuration file *great_expectations.yml* that *great_expectations init* command creates.

```
perform_action_list_operator:  # this is the name you will use when you invoke the␣
↪operator
  class_name: ActionListValidationOperator

  # the operator will call the following actions on each validation result
  # you can remove or add actions to this list. See the details in the actions
  # reference
```

(continues on next page)

(continued from previous page)

```
  action_list:
    - name: store_validation_result
      action:
        class_name: StoreValidationResultAction
        target_store_name: validations_store
    - name: send_slack_notification_on_validation_result
      action:
        class_name: SlackNotificationAction
        # put the actual webhook URL in the uncommitted/config_variables.yml file
        slack_webhook: ${validation_notification_slack_webhook}
        notify_on: all # possible values: "all", "failure", "success"
        renderer:
          module_name: great_expectations.render.renderer.slack_renderer
          class_name: SlackRenderer
    - name: update_data_docs
      action:
        class_name: UpdateDataDocsAction
```

Invocation

This is an example of invoking an instance of a Validation Operator from Python:

```
results = context.run_validation_operator(
    assets_to_validate=[batch0, batch1, ...],
    run_id="some_string_that_uniquely_identifies_this_run",
    validation_operator_name="perform_action_list_operator",
)
```

- *assets_to_validate* - an iterable that specifies the data assets that the operator will validate. The members of the list can be either batches or triples that will allow the operator to fetch the batch: (data_asset_name, expectation_suite_name, batch_kwargs) using this method: *get_batch()*
- run_id - pipeline run id, a timestamp or any other string that is meaningful to you and will help you refer to the result of this operation later
- validation_operator_name you can instances of a class that implements a Validation Operator

The *run* method returns an object that looks like this:

```
{
    'success: True/False, (True if all validations are successful)
    'details': {
        great_expectations.data_context.types.ExpectationSuiteIdentifier:
            {
            'validation_result': :ref:validation_result
            'actions_results':
                {
                'action_0_name': action result object (defined by the action),
                'action_1_name': action result object (defined by the action),
                ...
                'action_n_name`: action result object (defined by the action)
                }
            }
    }
}
```

WarningAndFailureExpectationSuitesValidationOperator

WarningAndFailureExpectationSuitesValidationOperator implements a business logic pattern that many data practitioners consider useful - grouping the expectations about a data asset into two expectation suites.

The "failure" expectation suite contains expectations that are considered important enough to stop the pipeline when they are violated. The rest of the expectations go into the "warning" expectation suite.

WarningAndFailureExpectationSuitesValidationOperator retrieves the two expectation suites ("failure" and "warning") for every data asset in the *assets_to_validate* argument of its *run* method. It does not require both suites to be present.

The operator invokes a list of actions on every validation result. The list is configured for the operator. Each action in the list must be an instance of ValidationAction class (or its descendants). Read more about actions here: *Actions*.

After completing all the validations, it sends a Slack notification with the success status.

Configuration

Below is an example of this operator's configuration:

```
run_warning_and_failure_expectation_suites:
    class_name: WarningAndFailureExpectationSuitesValidationOperator

    # the following two properties are optional - by default the operator looks for
    # expectation suites named "failure" and "warning".
    # You can use these two properties to override these names.
    # e.g., with expectation_suite_name_prefix=boo_ and
    # expectation_suite_name_suffixes = ["red", "green"], the operator
    # will look for expectation suites named "boo_red" and "boo_green"
    expectation_suite_name_prefix="",
    expectation_suite_name_suffixes=["failure", "warning"],

    # optional - if true, the operator will stop and exit after first failed␣
↪validation. false by default.
    stop_on_first_error=False,

    # put the actual webhook URL in the uncommitted/config_variables.yml file
    slack_webhook: ${validation_notification_slack_webhook}
    # optional - if "all" - notify always, "success" - notify only on success,
↪"failure" - notify only on failure
    notify_on="all"

    # the operator will call the following actions on each validation result
    # you can remove or add actions to this list. See the details in the actions
    # reference
    action_list:
      - name: store_validation_result
        action:
          class_name: StoreValidationResultAction
          target_store_name: validations_store
      - name: store_evaluation_params
        action:
          class_name: StoreEvaluationParametersAction
          target_store_name: evaluation_parameter_store
```

Invocation

This is an example of invoking an instance of a Validation Operator from Python:

```
results = context.run_validation_operator(
    assets_to_validate=[batch0, batch1, ...],
    run_id="some_string_that_uniquely_identifies_this_run",
    validation_operator_name="operator_instance_name",
)
```

- *assets_to_validate* - an iterable that specifies the data assets that the operator will validate. The members of the list can be either batches or triples that will allow the operator to fetch the batch: (data_asset_name, expectation_suite_name, batch_kwargs) using this method: *get_batch()*
- run_id - pipeline run id, a timestamp or any other string that is meaningful to you and will help you refer to the result of this operation later
- validation_operator_name you can instances of a class that implements a Validation Operator

The *run* method returns a result object.

The value of "success" is True if no critical expectation suites ("failure") failed to validate (non-critial warning") expectation suites are allowed to fail without affecting the success status of the run.

```
{
    "data_asset_identifiers": list of data asset identifiers
    "success": True/False,
    "failure": {
        expectation suite identifier: {
            "validation_result": validation result,
            "action_results": {action name: action result object}
        }
    }
    "warning": {
        expectation suite identifier: {
            "validation_result": validation result,
            "action_results": {action name: action result object}
        }
    }
}
```

Actions

An action is a way to take an arbitrary method and make it configurable and runnable within a data context.

The only requirement from an action is for it to have a take_action method.

SlackNotificationAction

SlackNotificationAction is a validation action that sends a Slack notification to a given webhook

Configuration

```
- name: send_slack_notification_on_validation_result
action:
  class_name: SlackNotificationAction
  # put the actual webhook URL in the uncommitted/config_variables.yml file
  slack_webhook: ${validation_notification_slack_webhook}
  notify_on: all # possible values: "all", "failure", "success"

  renderer:
    module_name: great_expectations.render.renderer.slack_renderer
    class_name: SlackRenderer
```

StoreValidationResultAction

StoreValidationResultAction is a namespace-aware validation action that stores a validation result in the store.

Configuration

```
- name: store_validation_result
action:
  class_name: StoreValidationResultAction
  # name of the store where the actions will store validation results
  # the name must refer to a store that is configured in the great_expectations.yml␣
↪file
  target_store_name: validations_store
```

StoreEvaluationParametersAction

StoreEvaluationParametersAction is a namespace-aware validation action that extracts evaluation parameters from a validation result and stores them in the store configured for this action.

Evaluation parameters allow expectations to refer to statistics/metrics computed in the process of validating other prior expectations.

Configuration

```
- name: store_evaluation_params
action:
  class_name: StoreEvaluationParametersAction
  # name of the store where the action will store the parameters
  # the name must refer to a store that is configured in the great_expectations.yml
↪file
  target_store_name: evaluation_parameter_store
```

UpdateDataDocsAction

UpdateDataDocsAction is a validation action that notifies the site builders of all the data docs sites of the data context that a validation result should be added to the data docs.

Configuration

```
- name: update_data_docs
action:
  class_name: UpdateDataDocsAction
  # this action has no additional configuration properties
```

Dependencies

When configured inside action_list of an operator, StoreValidationResultAction action has to be configured before this action, since the building of data docs fetches validation results from the store.

7.5.6 Usage Statistics

We use CDN fetch rates to get a sense of total community usage of Great Expectations. Specifically, we host images and style sheets on a public CDN and count the number of unique IPs from which resources are fetched.

Other than standard web request data, we don't collect any data data that could be used to identify individual users. You can suppress the images by changing `static_images_dir` in `great_expectations/render/view/templates/top_navbar.j2`.

Please reach out on Slack if you have any questions or comments.

7.5.7 Metrics Reference

Metrics are still a **beta feature** in Great Expectations. Expect changes to the API.

A Metric is a value that Great Expectations can use to evaluate expectations or to store externally. A metric could be a statistic, such as the minimum value of the column, or a more complex object, such as a histogram.

Expectation Validation Results and Expectation Suite Validation Results can expose metrics that are defined by specific expectations that have been validated, called "Expectation Defined Metrics." In the future, we plan to allow Expectations to have more control over the metrics that they generate, expose, and use in testing.

The following examples demonstrate how metrics are defined:

```
res = df.expect_column_values_to_be_in_set("Sex", ["male", "female"])
res.get_metric("expect_column_values_to_be_in_set.result.missing_count", column="Sex")
```

Last updated: |lastupdate|

7.6 Extending Great Expectations

7.6.1 Extending Great Expectations

It is possible to extend behavior in great expectations in several ways, described below.

Additional backends

Expectations require backend implementations to translate the semantically-meaningful expectation to the actual metrics and compute tasks required to evaluate it.

Inheriting Documentation

Expectations should maintain the same API and behavior independently of the backend that implements them. When implementing an expectation defined in the base *Dataset* for a new backend, we recommend that you add the *@DocInherit* decorator first to use the default dataset documentation for the expectation. That can help users of your dataset see consistent documentation no matter which backend is implementing the great_expectations API.

@DocInherit overrides your function's __get__ method with one that will replace the local docstring with the docstring from its parent. It is defined in *Dataset.util*.

7.6.2 Contributing

We welcome contributions to GE, and are eager both to continue to develop the library itself, as well as the broader ecosystem including plugins and examples using GE.

For contributing directly to great expectations, the contributors' guide is located here.

7.6.3 Improve documentation for Great Expectations

Contributing to documentation is a great way to help the community and get your feet wet as an open source contributor.

If you see a typo/mistake/gap anywhere in the Great Expectation documentation, a quick PR would be much appreciated.

Style guide for documentation

Note: as of 8/1/2019, this style guide is aspirational. It's applied unevenly across the repo. Look for this to tighten up a lot as we refine docs over the next couple of months.

Basics

- *The project name "Great Expectations" is always spaced and capitalized.* Good: "Great Expectations"; bad: "great_expectations", "great expectations", "GE."
- *Headers are capitalized like sentences.* Good: "Installing within a project." Bad: "Installing Within a Project."
- *We refer to ourselves in the first person plural*. Good: "we", "our". Bad: "I" . This helps us avoid awkward passive sentences. Occassionally, we refer to ourselves as "the Great Expectations team" (or community) for clarity.
- *We refer to developers and users as "you"*: "you can," "you should," "you might want to."
- *Core concepts are always capitalized.* Pretend the docs are a fantasy novel, and core concepts are magic. Good: "Like assertions in traditional python unit tests, Expectations provide a flexible, declarative language for describing expected behavior."
- *Core concepts are linked on first reference within each page.*
- *Class names are written in upper camel case and linked on first reference.* Good: "ValidationOperator." Bad: "validationOperator", "validation operator". If a word is both a core concept and a class name, link to the core concept unless the text refers specifically to the class.

Organization

Within the table of contents, each section has specific role to play.

- *Introduction* explains the Why of Great Expectations, so that potential users can quickly decide whether or not the library can help them.
- *Getting started* helps users get started quickly. Along the way it briefly orients new users to concepts that will be important to learn later.
- *Community* helps expand the Great Expectations community by explaining how to get in touch to ask questions, make contributions, etc.
- *Core concepts* are always phrased as nouns. These docs provide more examples of usage, and deeper explanations for why Great Expectations is set up the way it is.
- *reference* are always phrased as verbs: "Creating custom Expectations", "Deploying Great Expectations in Spark", etc. They help users accomplish specific goals that go beyond the generic Getting Started tutorials.
- *Changelog and roadmap*
- *Module docs*

CLI

The *CLI* has some conventions of its own.

- The CLI never writes to disk without asking first.
- Questions are always phrased as conversational sentences.
- Sections are divided by headers: "========== Profiling =========="
- We use punctuation: Please finish sentences with periods, questions marks, or an occasional exclamation point.
- Keep indentation and line spacing consistent! (We're pythonistas, natch.)
- Include exactly one blank line after every question.

- Within those constraints, shorter is better. When in doubt, shorten.
- Clickable links (usually to documentation) are blue.
- Copyable bash commands are green.
- All top-level bash commands must be nouns: "docs build", not "build docs"

Resources

- We follow the Sphinx guide for sections.

7.7 Supporting Resources

7.7.1 Supporting Resources

Great Expectations requires a python compute environment and access to data, either locally or through a database or distributed cluster. In addition, developing with great expectations relies heavily on tools in the Python engineering ecosystem: pip, virtual environments, and jupyter notebooks. We also assume some level of familiarity with git and version control.

See the links below for good, practical tutorials for these tools.

7.7.2 Tools Reference

pip

- https://pip.pypa.io/en/stable/
- https://www.datacamp.com/community/tutorials/pip-python-package-manager

virtual environments

- https://virtualenv.pypa.io/en/latest/
- https://python-guide-cn.readthedocs.io/en/latest/dev/virtualenvs.html
- https://www.dabapps.com/blog/introduction-to-pip-and-virtualenv-python/

jupyter notebooks and jupyter lab

- https://jupyter.org/
- https://jupyterlab.readthedocs.io/en/stable/
- https://towardsdatascience.com/jupyter-lab-evolution-of-the-jupyter-notebook-5297cacde6b

git

- https://git-scm.com/
- https://reference.github.com/
- https://www.atlassian.com/git/tutorials

7.7.3 Installing from github

If you plan to make changes to great expectations or live on the bleeding edge, you may want to clone from GitHub and pip install using the –editable flag.

```
$ git clone https://github.com/great-expectations/great_expectations.git
$ pip install -e great_expectations/
```

7.7.4 Great Expectations Glossary

Expectations

Expectations are assertions for data. They help accelerate data engineering and increase analytic integrity, by making it possible to answer a critical question:

- What can I expect of my data?

Expectations are declarative statements that a computer can evaluate, and that are semantically meaningful to humans, like expect_column_values_to_be_unique or expect_column_mean_to_be_between.

Expectation Configurations describe specific Expectations for data. They combine an Expectation and specific parameters to make it possible to evaluate whether the expectation is true on a dataset. For example, they might provide expected values or the name of a column whose values should be unique.

Expectation Suites combine multiple Expectation Configurations into an overall description of a dataset. Expectation Suites should have names corresponding to the kind of data they define, like "NPI" for National Provider Identifier data or "company.users" for a users table.

DataAssets and Validation

In addition to specifying Expectations, Great Expectations also allows you to validate your data against an Expectation Suite. Validation produces a detailed report of how the data meets your expectations – and where it doesn't.

DataAssets and Validations, answer the questions:

- How do I describe my Expectations to Great Expectations?
- Does my data meet my Expectations?

A **DataAsset** is a Great Expectations object that can create and validate Expectations against specific data. DataAssets are connected to data. They can evaluate Expectations wherever you access your data, using Pandas, Spark, or SqlAlchemy.

An •Expectation Validation Result** captures the output of checking an expectation against data. It describes whether the data met the expectation, and additional metrics from the data such as the percentage of unique values or observed mean.

An **Expectation Suite Validation Result** combines multiple Expectation Validation Results and metadata about the validation into a single report.

Datasources and Batches

Great Expectations lets you focus on your data, not writing tests. It validates your expectations no matter where the data is located.

Datasources, Generators, Batch Parameters and Batch Kwargs make it easier to connect Great Expectations to your data. Together, they address questions such as:

- How do I get data into my Great Expectations data asset?
- How do I tell my Datasource how to access my specific data?
- How do I use Great Expectations to store Batch Kwargs configurations or logically describe data when I need to build

equivalent Batch Kwargs for different datasources? - How do I know what data is available from my datasource?

A **Datasource** is a connection to a compute environment (a backend such as Pandas, Spark, or a SQL-compatible database) and one or more storage environments. It produces batches of data that Great Expectations can validate in that environment.

Batch Kwargs are specific instructions for a Datasource about what data should be prepared as a "batch" for validation. The batch could be a specific database table, the most recent log file delivered to S3, or even a subset of one of those objects such as the first 10,000 rows.

Batch Parameters provide instructions for how to retrieve stored Batch Kwargs or build new Batch Kwargs that reflect partitions, deliveries, or slices of logical data assets.

A **Batch Kwargs Generator** translates Batch Parameters to datasource-specific Batch Kwargs. A Batch Kwargs Generator can also identify data assets and partitions by inspecting a storage environment.

Profiling

Profiling helps you understand your data by describing it and even building expectation suites based on previous batches of data. Profiling lets you ask:

- What is this dataset like?

A **Profiler** reviews data assets and produces new Expectation Suites and Expectation Suite Validation Results that describe the data. A profiler can create a "stub" of high-level expectations based on what it sees in the data. Profilers can also be extended to create more specific expectations based on team conventions or statistical properties. Finally, Profilers can take advantage of metrics produced by Great Expectations when validating data to create useful overviews of data.

Data Docs

With Great Expectations, your tests can update your docs, and your docs can validate your data. Data Docs makes it possible to produce clear visual descriptions of what you expect, what you observe, and how they differ. Does my data meet my expectations?

An **Expectation Suite Renderer** creates a page that shows what you expect from data. Its language is prescriptive, for example translating a fully-configured expect_column_values_to_not_be_null expectation into "column "address" values must not be null, at least 80% of the time"

A **Validation Result Renderer** produces an overview of the result of validating a batch of data with an Expectation Suite. It shows the difference between observed and expected values.

A **Profiling Renderer** details the observed metrics produced from a validation without comparing them to specific expected values. It provides a detailed look into what Great Expectations learned about your data.

Data Context

A **Data Context** stitches together all the features available with Great Expectations, making it possible to easily manage configurations for datasources, and data docs sites and to store expectation suites and validations. Data Contexts also unlock more powerful features such as Evaluation Parameter Stores.

A **Data Context Configuration** is a yaml file that can be committed to source control to ensure that all the settings related to your validation are appropriately versioned and visible to your team. It can flexibly describe plugins and other customizations for accessing datasources or building data docs sites.

A **Store** allows you to manage access to Expectations, Validations and other Great Expectations assets in a standardized way, making it easy to share resources across a team that uses AWS, Azure, GCP, local storage, or something else entirely.

A **Metric** is simply a value produced by Great Expectations when evaluating one or more batches of data, such as an observed mean or distribution of data.

An **Evaluation Parameter** Store makes it possible to build expectation suites that depend on values from other batches of data, such as ensuring that the number of rows in a downstream dataset equals the number of unique values from an upstream one. A Data Context can manage a store to facilitate that validation scenario.

Validation Operators

A **Validation Operator** stitches together resources provided by the Data Context to build mini-programs that demonstrate the full potential of Great Expectations. They take configurable Actions such as updating Data Docs, sending a notification to your team about validation results, or storing a result in a shared S3 bucket.

7.7.5 Check Assets Pattern

While Great Expectations has nearly *50 built in expectations*, the need for complex data assertions is common.

Check assets are a **simple design pattern** that enables even more complex and fine-grained data tests, such as:

assertions on slices of the data For example: How many visits occurred last week?

assertions across logical tables For example: Does my visits table join with my patients table?

A check asset is a slice of data that is only created for validation purposes and may not feed into pipeline or analytical output.

These check assets should be built in your pipeline's native transformation language. For example, if your pipeline is primarily SQL, create an additional table or view that slices the data so that you can use the built in expectations found here: *Glossary of Expectations*.

Postgres SQL example.

Let's suppose we have a `visits` table and we want to make an assertion about the typical number of visits in the last 30 days.

1. Create a new table with an obvious name like `ge_asset_visits_last_30_days` that is populated with the following query:

```
SELECT *
FROM visits
WHERE visit_date > current_date - interval '30' day;
```

2. Create a new expectation suite against this new table. Again, we recommend using an obvious name such as `visits_last_30_days`.

3. Add an expectation as follows:

```
batch.expect_table_row_count_to_be_between(min_value=2000, max_value=5000)
```

CHAPTER

EIGHT

CONTRIBUTING

Welcome to the Great Expectations project! We're very glad you want to help out by contributing.

Our goal is to make your experience as great as possible. Please follow these steps to contribute:

Go to greatexpectations.io/slack and introduce yourself in the `#contributors` channel.

Follow *those instructions* to set up your dev environment.

Alternatively, for small changes that don't need to be tested locally (e.g. documentation changes), you can *make changes directly through Github*.

Issues in GitHub are a great place to start. Check out the help wanted and good first issue labels. Comment to let everyone know you're working on it.

If there's no issue for what you want to work on, please create one. Add a comment to let everyone know that you're working on it. We prefer small, incremental commits, because it makes the thought process behind changes easier to review.

When your changes are ready, run through our *Contribution checklist* for pull requests.

Note that if it's your first contribution, there is a standard *Contributor license agreement (CLA)* to sign.

Table of contents for Contributing:

8.1 Setting up your dev environment

8.1.1 Prerequisites

In order to contribute to Great Expectations, you will need the following:

- A GitHub account—this is sufficient if you *only want to contribute to the documentation*.
- If you want to contribute code, you will also need a working version of Git on your computer. Please refer to the Git setup instructions for your environment.
- We also recommend going through the SSH key setup process on GitHub for easier authentication.

8.1.2 Fork and clone the repository

1. Fork the Great Expectations repo

- Go to the Great Expectations repo on GitHub.
- Click the `Fork` button in the top right. This will make a copy of the repo in your own GitHub account.
- GitHub will take you to your forked version of the repository.

2. Clone your fork

- Click the green `Clone` button and choose the SSH or HTTPS URL depending on your setup.
- Copy the URL and run `git clone <url>` in your local terminal.
- This will clone the `develop` branch of the great_expectations repo. Please use `develop` (not `master`!) as the starting point for your work.
- Atlassian has a nice tutorial for developing on a fork.

3. Add the upstream remote

- On your local machine, cd into the `great_expectations` repo you cloned in the previous step.
- Run: `git remote add upstream git@github.com:great-expectations/great_expectations.git`
- This sets up a remote called `upstream` to track changes to the main branch.

4. Create a feature branch to start working on your changes.

- Ex: `git checkout -b feature/my-feature-name`
- We do not currently follow a strict naming convention for branches. Please pick something clear and self-explanatory, so that it will be easy for others to get the gist of your work.

8.1.3 Install python dependencies

5. Create a new virtual environment

- Make a new virtual environment (e.g. using virtualenv or conda), name it "great_expectations_dev" or similar.
- Ex: `virtualenv great_expectations_dev; source great_expectations_dev/bin/activate`
- This is not required, but highly recommended.

6. Install dependencies from requirements-dev.txt

- `pip install -r requirements-dev.txt`
- This will ensure that sure you have the right libraries installed in your python environment.

7. Install great_expectations from your cloned repo

- `pip install -e .`
- `-e` will install Great Expectations in "editable" mode. This is not required, but is often very convenient as a developer.

8.1.4 (Optional) Configure resources for testing and documentation

Depending on which features of Great Expectations you want to work on, you may want to configure different backends for local testing, such as postgresql and Spark. Also, there are a couple of extra steps if you want to build documentation locally.

If you want to develop against local postgresql:

- To simplify setup, the repository includes a docker-compose file that can stand up a local postgresql container. To use it, you'll need to have docker installed.
- Navigate to `assets/docker/postgresql` in your `great_expectations` repo and run `docker-compose up -d`
- Within the same directory, you can run `docker-compose ps` to verify that the container is running. You should see something like:

```
        Name                       Command              State          ␣
↪Ports
-------------------------------------------------------------------------------------
postgresql_travis_db_1   docker-entrypoint.sh postgres   Up      0.0.0.
↪0:5432->5432/tcp
```

- Once you're done testing, you can shut down your postgesql container by running `docker-compose down` from the same directory.
- Caution: If another service is using port 5432, docker may start the container but silently fail to set up the port. In that case, you will probably see errors like this:

```
psycopg2.OperationalError: could not connect to server: Connection refused
    Is the server running on host "localhost" (::1) and accepting
    TCP/IP connections on port 5432?
could not connect to server: Connection refused
    Is the server running on host "localhost" (127.0.0.1) and accepting
    TCP/IP connections on port 5432?
```

- Or this...

```
sqlalchemy.exc.OperationalError: (psycopg2.OperationalError) FATAL: ␣
↪database "test_ci" does not exist
(Background on this error at: http://sqlalche.me/e/e3q8)
```

If you want to develop against local Spark:

- In most cases, `pip install requirements-dev.txt` should set up pyspark for you.
- If you don't have Java installed, you will probably need to install it and set your `PATH` or `JAVA_HOME` environment variables appropriately.
- You can find official installation instructions for spark here.

If you want to build documentation locally:

- `pip install -r docs/requirements.txt`
- To build documentation, the command is `cd docs; make html`
- Documentation will be generated in `docs/build/html/` with the `index.html` as the index page.

8.1.5 Run tests to confirm that everything is working

You can run all tests by running `pytest` in the great_expectations directory root. Please see *Testing* for testing options and details.

8.1.6 Start coding!

At this point, you have everything you need to start coding!

8.2 Contribution checklist

Following these instructions helps us make sure the code review and merge process go smoothly.

8.2.1 Before submitting a pull request

Once your code is ready, please go through the following checklist before submitting a pull request.

1. Have you signed the CLA?

- A Contributor License Agreement helps guarantee that contributions to Great Expectations will always remain free and open.
- Please see *Contributor license agreement (CLA)* for more information and instructions for how to sign the CLA the first time you contriubte to Great Expectations.
- If you've included your (physical) mailing address in the CLA, we'll send you a personalized Great Expectations mug once your first PR is merged!

2. Have you followed the Style Guide for code and comments?

- The *Style Guide* is here.
- Thanks for helping us keep the codebase and documentation clean and consistent, so that it's easier to maintain it as a community!

3. Is your branch up to date with upstream/develop?

- Update your local repository with the most recent code from the main Great Expectations repository.
- For changes with few or no merge conflicts, you can do this by creating a draft pull request in GitHub and clicking `Update branch`.
- You can also rebase your branch from `upstream/develop`. In general, the steps are:
  1. Run `git fetch upstream` then `git rebase upstream/develop`.
  2. Fix any merge conflicts that arise from the rebase.
  3. Make sure to add and commit all your changes in this step.
  4. Re-run tests to ensure the rebase did not introduce any new issues.
- Atlassian and Github both have good tutorials for rebasing: Atlassian's tutorial, Github's tutorial.

4. Have you written and run all the tests you need?

- See *Writing unit and integration tests* for details on how to write unit tests in Great Expectations.
- Please make certain to run `pytest` to verify that all tests pass locally. See *Running tests* for details.

5. Have you added a bullet with your changes under the "develop" heading in the Changelog?

- Please add a bullet point to `docs/changelog/changelog.rst`, in the `develop` section.
- You can see the past Changelog here: *Changelog*

If you've checked off all these items, you're now ready to submit a pull request!

8.2.2 How to submit a pull request

When you're done with your work...

1. Create a PR

- Push to the remote fork of your repo.
- Follow these instructions to create a PR from your commit.
- In the PR, choose a short title which sums up the changes that you have made, and in the body provide more details about what your changes do. Also mention the number of the issue where discussion has taken place, e.g. "Closes #123".

2. Confirm the contributor license agreement (CLA)

- If you've followed the checklist above, you will have already signed the CLA and won't see the CLA bot.
- Otherwise, you will see a comment from the "CLA Bot" on the PR that asks you to complete the CLA form. Please do so.
- Once you've signed the form, add a new comment to the PR with the line `@cla-bot check`. This will trigger the CLA bot to refresh.

3. Verify continuous integration checks

- Wait for the other continuous integration (CI) checks to go green and watch out for a comment from the automated linter that checks for syntax and formatting issues.
- Fix any issues that are flagged.

4. Wait for a core team member to approve and merge your PR

- Once all checks pass, a Great Expectations team member will approve your PR and merge it.
- GitHub will notify you of comments or a successful merge according to your notification settings.

5. Resolve any issues

- There will probably be discussion about the pull request. It's normal for a request to require some changes before merging it into the main Great Expectations project. We enjoy working with contributors to help them get their code accepted. There are many approaches to fixing a problem and it is important to find the best approach before writing too much code!

6. Do a victory dance

- Congratulations! You've just contributed to Great Expectations!

8.3 Making changes directly through Github

If you want to change documentation, but not code, we suggest using the GitHub markdown editor, which means you don't have to fork the repo at all. Here's how you do this:

8.3.1 Start editing

1A. Edit from docs.greatexpecations.io

- Go to the Great Expectations docs.
- On each page, you'll see an `Edit on GitHub` button in the top right. Click this to go to the source file in the Great Expectations GitHub repo.

1B. Edit from Github

- If you're already on GitHub, the docs are located in `great_expectations > docs`. You can directly navigate to the respective page you want to edit (but getting there from docs.great_expectations.io is a little easier).
- In the top right of the grey header bar of the actual file, click the pencil icon to get into edit mode on GitHub.

2. Make edits

- Make your edits and use the Preview tab to preview changes.
- Please pay close attention to the *Style Guide*.

8.3.2 Submit a pull request

3. Submit your edits as a PR

- When you're done, add a meaningful commit message at the bottom. Use a short title and a meaningful explanation of what you changed and why.
- Click the `Propose File Change` button at the bottom of the page.
- Click the `Create Pull Request` button.
- Optionally: Add comment to explain your change, if it's not already in the commit message.
- Click the next `Create Pull Request` button to create the actual PR.

4. Sign the CLA

- If this is your first contribution to Great Expectations, You will see a comment from the "CLA Bot" that asks you to complete the Contributor Licence Agreement form.
- Please complete the form and comment on the PR to say that you've signed the form.

5. Verify continuous integration checks

- Wait for the other Continuous Integration (CI) checks to go green and watch out for a comment from the automated linter that checks for syntax and formatting issues.
- Fix any issues that are flagged. (For documentation changes, it's unlikely that you'll have any issues.)

6. Wait for a core team member to approve and merge your PR

- Once all checks pass, a Great Expectations team member will approve your PR and merge it.
- GitHub will notify you of comments or a successful merge according to your notification settings.

- If there are any issues, please address them promptly.

7. Do a victory dance

- Congratulations! You've just contributed to Great Expectations!

8.4 Testing

8.4.1 Running tests

You can run all unit tests by running `pytest` in the `great_expectations` directory root.

If you did not configure optional backends for testing, tests against these backends will fail.

You can suppress these tests by adding the following flags:

- `--no-postgresql` will skip postgres tests
- `--no-spark` will skip spark tests
- `--no-sqlalchemy` will skip all tests using sqlalchemy (i.e. all database backends)

For example, you can run `pytest --no-spark --no-sqlalchemy` to skip all local backend tests (with the exception of the pandas backend). Please note that these tests will still be run by the CI as soon as you open a PR, so some tests might fail if your code changes affected them.

Note: as of early 2020, the tests generate many warnings. Most of these are generated by dependencies (pandas, sqlalchemy, etc.) You can suppress them with pytest's `--disable-pytest-warnings` flag: `pytest --no-spark --no-sqlalchemy --disable-pytest-warnings`

8.4.2 Writing unit and integration tests

Production code in Great Expectations must be thoroughly tested. In general, we insist on unit tests for all branches of every method, including likely error states. Most new feature contributions should include several unit tests. Contributions that modify or extend existing features should include a test of the new behavior.

Experimental code in Great Expectations need only be tested lightly. We are moving to a convention where experimental features are clearly labeled in documentation and the code itself. However, this convention is not uniformly applied today.

Most of Great Expectations' integration testing is in the CLI, which naturally exercises most of the core code paths. Because integration tests require a lot of developer time to maintain, most contributions should *not* include new integration tests, unless they change the CLI itself.

Note: we do not currently test Great Expectations against all types of SQL database. CI test coverage for SQL is limited to postgresql and sqlite. We have observed some bugs because of unsupported features or differences in SQL dialects, and we are actively working to improve dialect-specific support and testing.

8.4.3 Unit tests for Expectations

One of Great Expectations' important promises is that the same Expectation will produce the same result across all supported execution environments: pandas, sqlalchemy, and Spark.

To accomplish this, Great Expectations encapsulates unit tests for Expectations as JSON files. These files are used as fixtures and executed using a specialized test runner that executes tests against all execution environments.

Test fixture files are structured as follows:

```
{
    "expectation_type" : "expect_column_max_to_be_between",
    "datasets" : [{
        "data" : {...},
        "schemas" : {...},
        "tests" : [...]
    }]
}
```

Each item under `datasets` includes three entries: `data`, `schemas`, and `tests`.

data

> ...defines a dataframe of sample data to apply Expectations against. The dataframe is defined as a dictionary of lists, with keys containing column names and values containing lists of data entries. All lists within a dataset must have the same length.

```
"data" : {
    "w" : [1, 2, 3, 4, 5, 5, 4, 3, 2, 1],
    "x" : [2, 3, 4, 5, 6, 7, 8, 9, null, null],
    "y" : [1, 1, 1, 2, 2, 2, 3, 3, 3, 4],
    "z" : ["a", "b", "c", "d", "e", null, null, null, null, null],
    "zz" : ["1/1/2016", "1/2/2016", "2/2/2016", "2/2/2016", "3/1/2016", "2/1/
↪2017", null, null, null, null],
    "a" : [null, 0, null, null, 1, null, null, 2, null, null],
},
```

schemas

> ...define the types to be used when instantiating tests against different execution environments, including different SQL dialects. Each schema is defined as dictionary with column names and types as key-value pairs. If the schema isn't specified for a given execution environment, Great Expectations will introspect values and attempt to guess the schema.

```
"schemas": {
    "sqlite": {
        "w" : "INTEGER",
        "x" : "INTEGER",
        "y" : "INTEGER",
        "z" : "VARCHAR",
        "zz" : "DATETIME",
        "a" : "INTEGER",
    },
    "postgresql": {
        "w" : "INTEGER",
        "x" : "INTEGER",
        "y" : "INTEGER",
        "z" : "TEXT",
        "zz" : "TIMESTAMP",
```

(continues on next page)

(continued from previous page)

```
        "a" : "INTEGER",
    }
},
```

tests

...define the tests to be executed against the dataframe. Each item in `tests` must have `title`, `exact_match_out`, `in`, and `out`. The test runner will execute the named Expectation once for each item, with the values in `in` supplied as kwargs.

The test passes if the values in the expectation validation result correspond with the values in `out`. If `exact_match_out` is true, then every field in the Expectation output must have a corresponding, matching field in `out`. If it's false, then only the fields specified in `out` need to match. For most use cases, false is a better fit, because it allows narrower targeting of the relevant output.

`suppress_test_for` is an optional parameter to disable an Expectation for a specific list of backends.

See an example below. For other examples

```
"tests" : [{
    "title": "Basic negative test case",
    "exact_match_out" : false,
    "in": {
        "column": "w",
        "result_format": "BASIC",
        "min_value": null,
        "max_value": 4
    },
    "out": {
        "success": false,
        "observed_value": 5
    },
    "suppress_test_for": ["sqlite"]
},
...
]
```

The test fixture files are stored in subdirectories of `tests/test_definitions/` corresponding to the class of Expectation:

- column_map_expectations
- column_aggregate_expectations
- column_pair_map_expectations
- column_distributional_expectations
- multicolumn_map_expectations
- other_expectations

By convention, the name of the the file is the name of the Expectation, with a `.json` suffix. Creating a new json file will automatically add the new Expectation tests to the test suite.

Note: If you are implementing a new Expectation, but don't plan to immediately implement it for all execution environments, you should add the new test to the appropriate list(s) in the *candidate_test_is_on_temporary_notimplemented_list* method within *tests/test_utils.py*. Often, we see Expectations developed first for pandas, then later extended to SqlAlchemy and Spark.

You can run just the Expectation tests with `pytest tests/test_definitions/test_expectations.py`.

8.4.4 Manual testing

We do manual testing (e.g. against various databases and backends) before major releases and in response to specific bugs and issues.

8.5 Style Guide

- Ensure any new features or behavioral differences introduced by your changes are documented in the docs, and ensure you have docstrings on your contributions. We use the Sphinx's Napoleon extension to build documentation from Google-style docstrings.
- Avoid abbreviations, e.g. use *column_index* instead of *column_idx*.
- Use unambiguous expectation names, even if they're a bit longer, e.g. use *expect_columns_to_match_ordered_list* instead of *expect_columns_to_be*.
- Expectation names should reflect their decorators:
  - *expect_table_...* for methods decorated directly with *@expectation*
  - *expect_column_values_...* for *@column_map_expectation*
  - *expect_column_...* for *@column_aggregate_expectation*
  - *expect_column_pair_values...* for *@column_pair_map_expectation*

These guidelines should be followed consistently for methods and variables exposed in the API. They aren't intended to be strict rules for every internal line of code in every function.

8.6 Miscellaneous

8.6.1 Core team

- James Campbell
- Abe Gong
- Eugene Mandel
- Rob Lim
- Taylor Miller
- Alex Shertinsky
- Tal Gluck
- Kyle Eaton
- Sam Bail
- Priya Joseph
- Ben Castleton

8.6.2 Contributor license agreement (CLA)

When you contribute code, you affirm that the contribution is your original work and that you license the work to the project under the project's open source license. Whether or not you state this explicitly, by submitting any copyrighted material via pull request, email, or other means you agree to license the material under the project's open source license and warrant that you have the legal authority to do so.

Please make sure you have signed our Contributor License Agreement (either Individual Contributor License Agreement v1.0 or Software Grant and Corporate Contributor License Agreement ("Agreement") v1.0).

We are not asking you to assign copyright to us, but to give us the right to distribute your code without restriction. We ask this of all contributors in order to assure our users of the origin and continuing existence of the code. You only need to sign the CLA once.

8.6.3 Release checklist

GE core team members use this checklist to ship releases.

- Merge all approved PRs into `develop`.
- Make a new branch from `develop` called something like `release-prep`.
- In this branch, update the version number in the `.travis.yml` file (look in the deploy section). (This sed snippet is handy if you change the numbers `sed -i '' 's/0\.9\.6/0\.9\.7/g' .travis.yml`)
- Update the `changelog.rst`: move all things under `develop` under a new heading with the new release number.
- Submit this as a PR against `develop`
- After successful checks, get it approved and merged.
- Update your local branches and switch to master: `git fetch --all; git checkout master; git pull`.
- Merge the now-updated `develop` branch into `master` and trigger the release: `git merge origin/develop; git push`
- Wait for all the builds to complete (including the deploy job).
- Check PyPI for the new release
- Create an annotated git tag:
  - Run `git tag -a <<VERSION>> -m "<<VERSION>>"` with the correct new version
  - Push the tag up by running `git push origin <<VERSION>>` with the correct new version
  - Merge `master` into `develop` so that the tagged commit becomes part of the history for `develop`: `git checkout develop; git pull; git merge master`
  - On develop, add a new "develop" section header to changelog.rst, and push the updated file with message "Update changelog for develop"
- Create the release on GitHub with the version number. Copy the changelog notes into the release notes, and update any rst-specific links to use github issue numbers.
- Notify kyle@superconductive.com about any community-contributed PRs that should be celebrated.
- Socialize the release on GE slack by copying the changelog with an optional nice personal message (thank people if you can)

CHAPTER

NINE

COMMUNITY

We're committed to supporting and growing the community around Great Expectations. It's not enough to build a great tool. We want to build a great community as well.

Open source doesn't always have the best reputation for being friendly and welcoming, and that makes us sad. Everyone belongs in open source, and Great Expectations is dedicated to making you feel welcome.

9.1 Get in touch with the Great Expectations team

Join our public slack channel here: join slack. We're not always available, but we're there and responsive an awful lot of the time.

9.2 Ask a question

Slack is good for that, too: join slack.

9.3 File a bug report or feature request

If you have bugfix or feature request, please upvote an existing issue or open a new issue on GitHub and we'll see what we can do.

9.4 Contribute code or documentation

We welcome contributions to Great Expectations. Please start with our *Contributing* guide and don't be shy with questions!

CHAPTER

TEN

ROADMAP AND CHANGELOG

10.1 Planned Features

- Improved variable typing
- Support for non-tabular datasources (e.g. JSON, XML, AVRO)
- Conditional expectations
- Multi-batch expectations

10.2 Changelog

10.2.1 develop

10.2.2 0.9.10

- Data Docs: improve configurability of site_section_builders
- TupleFilesystemStoreBackend now ignore *.ipynb_checkpoints* directories #1203
- bugfix for Data Docs links encoding on S3 #1235
- Add evaluation parameters support in WarningAndFailureExpectationSuitesValidationOperator #1284 thanks @balexander
- Fix compatibility with MS SQL Server. #1269 thanks @kepiej
- Bug fixes for query_generator #1292 thanks @ian-whitestone

10.2.3 0.9.9

- Allow evaluation parameters support in run_validation_operator
- Add log_level parameter to jupyter_ux.setup_notebook_logging.
- Add experimental display_profiled_column_evrs_as_section and display_column_evrs_as_section methods, with a minor (nonbreaking) refactor to create a new _render_for_jupyter method.
- Allow selection of site in UpdateDataDocsAction with new arg target_site_names in great_expectations.yml
- Fix issue with regular expression support in BigQuery (#1244)

10.2.4 0.9.8

- Allow basic operations in evaluation parameters, with or without evaluation parameters.
- When unexpected exceptions occur (e.g., during data docs rendering), the user will see detailed error messages, providing information about the specific issue as well as the stack trace.
- Remove the "project new" option from the command line (since it is not implemented; users can only run "init" to create a new project).
- Update type detection for bigquery based on driver changes in pybigquery driver 0.4.14. Added a warning for users who are running an older pybigquery driver
- added execution tests to the NotebookRenderer to mitigate codegen risks
- **Add option "persist", true by default, for SparkDFDataset to persist the DataFrame it is passed. This addresses #1133 in a**
  - Disabling this option should *only* be done if the user has *already* externally persisted the DataFrame, or if the dataset is too large to persist but *computations are guaranteed to be stable across jobs*.
- Enable passing dataset kwargs through datasource via dataset_options batch_kwarg.
- Fix AttributeError when validating expectations from a JSON file
- Data Docs: fix bug that was causing erratic scrolling behavior when table of contents contains many columns
- Data Docs: add ability to hide how-to buttons and related content in Data Docs

10.2.5 0.9.7

- **Update marshmallow dependency to >3. NOTE: as of this release, you MUST use marshamllow >3.0, which REQUIRES p**
  - Schema checking is now stricter for expectation suites, and data_asset_name must not be present as a top-level key in expectation suite json. It is safe to remove.
  - Similarly, datasource configuration must now adhere strictly to the required schema, including having any required credentials stored in the "credentials" dictionary.
- New beta CLI command: *tap new* that generates an executable python file to expedite deployments. (#1193) @Aylr
- bugfix in TableBatchKwargsGenerator docs
- Added feature maturity in README (#1203) @kyleaton
- Fix failing test that should skip if postgresql not running (#1199) @cicdw

10.2.6 0.9.6

- validate result dict when instantiating an ExpectationValidationResult (#1133)
- DataDocs: Expectation Suite name on Validation Result pages now link to Expectation Suite page
- *great_expectations init*: cli now asks user if csv has header when adding a Spark Datasource with csv file
- Improve support for using GCP Storage Bucket as a Data Docs Site backend (thanks @hammadzz)
- fix notebook renderer handling for expectations with no column kwarg and table not in their name (#1194)

10.2.7 0.9.5

- Fixed unexpected behavior with suite edit, data docs and jupyter
- pytest pinned to 5.3.5

10.2.8 0.9.4

- Update CLI *init* flow to support snowflake transient tables
- Use filename for default expectation suite name in CLI *init*
- Tables created by SqlAlchemyDataset use a shorter name with 8 hex characters of randomness instead of a full uuid
- Better error message when config substitution variable is missing
- removed an unused directory in the GE folder
- removed obsolete config error handling
- Docs typo fixes
- Jupyter notebook improvements
- *great_expectations init* improvements
- Simpler messaging in valiation notebooks
- replaced hacky loop with suite list call in notebooks
- CLI suite new now supports *–empty* flag that generates an empty suite and opens a notebook
- add error handling to *init* flow for cases where user tries using a broken file

10.2.9 0.9.3

- Add support for transient table creation in snowflake (#1012)
- Improve path support in TupleStoreBackend for better cross-platform compatibility
- **New features on *ExpecatationSuite***
  - *.add_citation()*
  - *get_citations()*
- *SampleExpectationsDatasetProfiler* now leaves a citation containing the original batch kwargs
- *great_expectations suite edit* now uses batch_kwargs from citations if they exist
- Bugfix :: suite edit notebooks no longer blow away the existing suite while loading a batch of data
- More robust and tested logic in *suite edit*
- DataDocs: bugfixes and improvements for smaller viewports
- Bugfix :: fix for bug that crashes SampleExpectationsDatasetProfiler if unexpected_percent is of type decimal.Decimal (#1109)

10.2.10 0.9.2

- Fixes #1095
- Added a *list_expectation_suites* function to *data_context*, and a corresponding CLI function - *suite list*.
- CI no longer enforces legacy python tests.

10.2.11 0.9.1

- Bugfix for dynamic "How to Edit This Expectation Suite" command in DataDocs

10.2.12 0.9.0

Version 0.9.0 is a major update to Great Expectations! The DataContext has continued to evolve into a powerful tool for ensuring that Expectation Suites can properly represent the way users think about their data, and upgrading will make it much easier to store and share expectation suites, and to build data docs that support your whole team. You'll get awesome new features including improvements to data docs look and the ability to choose and store metrics for building flexible data quality dashboards.

The changes for version 0.9.0 fall into several broad areas:

1. Onboarding

Release 0.9.0 of Great Expectations makes it much easier to get started with the project. The *init* flow has grown to support a much wider array of use cases and to use more natural language rather than introducing GreatExpectations concepts earlier. You can more easily configure different backends and datasources, take advantage of guided walkthroughs to find and profile data, and share project configurations with colleagues.

If you have already completed the *init* flow using a previous version of Great Expectations, you do not need to rerun the command. However, **there are some small changes to your configuration that will be required**. See *Migrating Between Versions* for details.

2. CLI Command Improvements

With this release we have introduced a consistent naming pattern for accessing subcommands based on the noun (a Great Expectations object like *suite* or *docs*) and verb (an action like *edit* or *new*). The new user experience will allow us to more naturally organize access to CLI tools as new functionality is added.

3. Expectation Suite Naming and Namespace Changes

Defining shared expectation suites and validating data from different sources is much easier in this release. The DataContext, which manages storage and configuration of expectations, validations, profiling, and data docs, no longer requires that expectation suites live in a datasource-specific "namespace." Instead, you should name suites with the logical name corresponding to your data, making it easy to share them or validate against different data sources. For example, the expectation suite "npi" for National Provider Identifier data can now be shared across teams who access the same logical data in local systems using Pandas, on a distributed Spark cluster, or via a relational database.

Batch Kwargs, or instructions for a datasource to build a batch of data, are similarly freed from a required namespace, and you can more easily integrate Great Expectations into workflows where you do not need to use a BatchKwargsGenerator (usually because you have a batch of data ready to validate, such as in a table or a known directory).

The most noticeable impact of this API change is in the complete removal of the DataAssetIdentifier class. For example, the *create_expectation_suite* and *get_batch* methods now no longer require a data_asset_name parameter, relying only on the expectation_suite_name and batch_kwargs to do their job. Similarly, there is no more asset name normalization required. See the upgrade guide for more information.

4. Metrics and Evaluation Parameter Stores

Metrics have received much more love in this release of Great Expectations! We've improved the system for declaring evaluation parameters that support dependencies between different expectation suites, so you can easily identify a particular field in the result of one expectation to use as the input into another. And the MetricsStore is now much more flexible, supporting a new ValidationAction that makes it possible to select metrics from a validation result to be saved in a database where they can power a dashboard.

5. Internal Type Changes and Improvements

Finally, in this release, we have done a lot of work under the hood to make things more robust, including updating all of the internal objects to be more strongly typed. That change, while largely invisible to end users, paves the way for some really exciting opportunities for extending Great Expectations as we build a bigger community around the project.

We are really excited about this release, and encourage you to upgrade right away to take advantage of the more flexible naming and simpler API for creating, accessing, and sharing your expectations. As always feel free to join us on Slack for questions you don't see addressed!

10.2.13 0.8.9__develop

10.2.14 0.8.8

- Add support for allow_relative_error to expect_column_quantile_values_to_be_between, allowing Redshift users access to this expectation
- Add support for checking backend type information for datetime columns using expect_column_min_to_be_between and expect_column_max_to_be_between

10.2.15 0.8.7

- Add support for expect_column_values_to_be_of_type for BigQuery backend (#940)
- Add image CDN for community usage stats
- Documentation improvements and fixes

10.2.16 0.8.6

- Raise informative error if config variables are declared but unavailable
- Update ExpectationsStore defaults to be consistent across all FixedLengthTupleStoreBackend objects
- Add support for setting spark_options via SparkDFDatasource
- Include tail_weights by default when using build_continuous_partition_object
- Fix Redshift quantiles computation and type detection
- Allow boto3 options to be configured (#887)

10.2.17 0.8.5

- BREAKING CHANGE: move all reader options from the top-level batch_kwargs object to a sub-dictionary called "reader_options" for SparkDFDatasource and PandasDatasource. This means it is no longer possible to specify supplemental reader-specific options at the top-level of *get_batch*, *yield_batch_kwargs* or *build_batch_kwargs* calls, and instead, you must explicitly specify that they are reader_options, e.g. by a call such as: *context.yield_batch_kwargs(data_asset_name, reader_options={'encoding': 'utf-8'})*.
- BREAKING CHANGE: move all query_params from the top-level batch_kwargs object to a sub-dictionary called "query_params" for SqlAlchemyDatasource. This means it is no longer possible to specify supplemental query_params at the top-level of *get_batch*, *yield_batch_kwargs* or *build_batch_kwargs* calls, and instead, you must explicitly specify that they are query_params, e.g. by a call such as: *context.yield_batch_kwargs(data_asset_name, query_params={'schema': 'foo'})*.
- Add support for filtering validation result suites and validation result pages to show only failed expectations in generated documentation
- Add support for limit parameter to batch_kwargs for all datasources: Pandas, SqlAlchemy, and SparkDF; add support to generators to support building batch_kwargs with limits specified.
- Include raw_query and query_params in query_generator batch_kwargs
- Rename generator keyword arguments from data_asset_name to generator_asset to avoid ambiguity with normalized names
- Consistently migrate timestamp from batch_kwargs to batch_id
- Include batch_id in validation results
- Fix issue where batch_id was not included in some generated datasets
- Fix rendering issue with expect_table_columns_to_match_ordered_list expectation
- Add support for GCP, including BigQuery and GCS
- Add support to S3 generator for retrieving directories by specifying the *directory_assets* configuration
- Fix warning regarding implicit class_name during init flow
- Expose build_generator API publicly on datasources
- Allow configuration of known extensions and return more informative message when SubdirReaderBatchKwargsGenerator cannot find relevant files.
- Add support for allow_relative_error on internal dataset quantile functions, and add support for build_continuous_partition_object in Redshift
- Fix truncated scroll bars in value_counts graphs

10.2.18 0.8.4.post0

- Correct a packaging issue resulting in missing notebooks in tarball release; update docs to reflect new notebook locations.

10.2.19 0.8.4

- Improved the tutorials that walk new users through the process of creating expectations and validating data
- Changed the flow of the init command - now it creates the scaffolding of the project and adds a datasource. After that users can choose their path.
- Added a component with links to useful tutorials to the index page of the Data Docs website
- Improved the UX of adding a SQL datasource in the CLI - now the CLI asks for specific credentials for Postgres, MySQL, Redshift and Snowflake, allows continuing debugging in the config file and has better error messages
- Added batch_kwargs information to DataDocs validation results
- Fix an issue affecting file stores on Windows

10.2.20 0.8.3

- Fix a bug in data-docs' rendering of mostly parameter
- Correct wording for expect_column_proportion_of_unique_values_to_be_between
- Set charset and meta tags to avoid unicode decode error in some browser/backend configurations
- Improve formatting of empirical histograms in validation result data docs
- Add support for using environment variables in *config_variables_file_path*
- Documentation improvements and corrections

10.2.21 0.8.2.post0

- Correct a packaging issue resulting in missing css files in tarball release

10.2.22 0.8.2

- Add easier support for customizing data-docs css
- Use higher precision for rendering 'mostly' parameter in data-docs; add more consistent locale-based formatting in data-docs
- Fix an issue causing visual overlap of large numbers of validation results in build-docs index
- Documentation fixes (thanks @DanielOliver!) and improvements
- Minor CLI wording fixes
- Improved handling of MySql temporary tables
- Improved detection of older config versions

10.2.23 0.8.1

- Fix an issue where version was reported as '0+unknown'

10.2.24 0.8.0

Version 0.8.0 is a significant update to Great Expectations, with many improvements focused on configurability and usability. See the *Migrating Between Versions* guide for more details on specific changes, which include several breaking changes to configs and APIs.

Highlights include:

1. Validation Operators and Actions. Validation operators make it easy to integrate GE into a variety of pipeline runners. They offer one-line integration that emphasizes configurability. See the *Validation Operators And Actions Introduction* feature guide for more information.
    - The DataContext *get_batch* method no longer treats *expectation_suite_name* or *batch_kwargs* as optional; they must be explicitly specified.
    - The top-level GE validate method allows more options for specifying the specific data_asset class to use.
2. First-class support for plugins in a DataContext, with several features that make it easier to configure and maintain DataContexts across common deployment patterns.
    - **Environments**: A DataContext can now manage *Managing Environment and Secrets* more easily thanks to more dynamic and flexible variable substitution.
    - **Stores**: A new internal abstraction for DataContexts, *Stores*, make extending GE easier by consolidating logic for reading and writing resources from a database, local, or cloud storage.
    - **Types**: Utilities configured in a DataContext are now referenced using *class_name* and *module_name* throughout the DataContext configuration, making it easier to extend or supplement pre-built resources. For now, the "type" parameter is still supported but expect it to be removed in a future release.
3. Partitioners: Batch Kwargs are clarified and enhanced to help easily reference well-known chunks of data using a partition_id. Batch ID and Batch Fingerprint help round out support for enhanced metadata around data assets that GE validates. See *Batch Identifiers* for more information. The *GlobReaderBatchKwargsGenerator*, *QueryBatchKwargsGenerator*, *S3GlobReaderBatchKwargsGenerator*, *SubdirReaderBatchKwargsGenerator*, and *TableBatchKwargsGenerator* all support partition_id for easily accessing data assets.
4. Other Improvements:
    - We're beginning a long process of some under-the-covers refactors designed to make GE more maintainable as we begin adding additional features.
    - Restructured documentation: our docs have a new structure and have been reorganized to provide space for more easily adding and accessing reference material. Stay tuned for additional detail.
    - The command build-documentation has been renamed build-docs and now by default opens the Data Docs in the users' browser.

10.2.25 v0.7.11

- Fix an issue where head() lost the column name for SqlAlchemyDataset objects with a single column
- Fix logic for the 'auto' bin selection of *build_continuous_partition_object*
- Add missing jinja2 dependency
- Fix an issue with inconsistent availability of strict_min and strict_max options on expect_column_values_to_be_between
- Fix an issue where expectation suite evaluation_parameters could be overriden by values during validate operation

10.2.26 v0.7.10

- Fix an issue in generated documentation where the Home button failed to return to the index
- Add S3 Generator to module docs and improve module docs formatting
- Add support for views to QueryBatchKwargsGenerator
- Add success/failure icons to index page
- Return to uniform histogram creation during profiling to avoid large partitions for internal performance reasons

10.2.27 v0.7.9

- Add an S3 generator, which will introspect a configured bucket and generate batch_kwargs from identified objects
- Add support to PandasDatasource and SparkDFDatasource for reading directly from S3
- Enhance the Site Index page in documentation so that validation results are sorted and display the newest items first when using the default run-id scheme
- Add a new utility method, *build_continuous_partition_object* which will build partition objects using the dataset API and so supports any GE backend.
- Fix an issue where columns with spaces in their names caused failures in some SqlAlchemyDataset and SparkDFDataset expectations
- Fix an issue where generated queries including null checks failed on MSSQL (#695)
- Fix an issue where evaluation parameters passed in as a set instead of a list could cause JSON serialization problems for the result object (#699)

10.2.28 v0.7.8

- BREAKING: slack webhook URL now must be in the profiles.yml file (treat as a secret)
- Profiler improvements: - Display candidate profiling data assets in alphabetical order - Add columns to the expectation_suite meta during profiling to support human-readable description information
- Improve handling of optional dependencies during CLI init
- Improve documentation for create_expectations notebook
- Fix several anachronistic documentation and docstring phrases (#659, #660, #668, #681; #thanks @StevenMMortimer)

- Fix data docs rendering issues: - documentation rendering failure from unrecognized profiled column type (#679; thanks @dinedal)) - PY2 failure on encountering unicode (#676)

10.2.29 0.7.7

- Standardize the way that plugin module loading works. DataContext will begin to use the new-style class and plugin identification moving forward; yml configs should specify class_name and module_name (with module_name optional for GE types). For now, it is possible to use the "type" parameter in configuration (as before).
- Add support for custom data_asset_type to all datasources
- Add support for strict_min and strict_max to inequality-based expectations to allow strict inequality checks (thanks @RoyalTS!)
- Add support for reader_method = "delta" to SparkDFDatasource
- Fix databricks generator (thanks @sspitz3!)
- Improve performance of DataContext loading by moving optional import
- Fix several memory and performance issues in SparkDFDataset. - Use only distinct value count instead of bringing values to driver - Migrate away from UDF for set membership, nullity, and regex expectations
- Fix several UI issues in the data_documentation - Move prescriptive dataset expectations to Overview section - Fix broken link on Home breadcrumb - Scroll follows navigation properly - Improved flow for long items in value_set - Improved testing for ValidationRenderer - Clarify dependencies introduced in documentation sites - Improve testing and documentation for site_builder, including run_id filter - Fix missing header in Index page and cut-off tooltip - Add run_id to path for validation files

10.2.30 0.7.6

- New Validation Renderer! Supports turning validation results into HTML and displays differences between the expected and the observed attributes of a dataset.
- Data Documentation sites are now fully configurable; a data context can be configured to generate multiple sites built with different GE objects to support a variety of data documentation use cases. See data documentation guide for more detail.
- CLI now has a new top-level command, *build-documentation* that can support rendering documentation for specified sites and even named data assets in a specific site.
- Introduced DotDict and LooselyTypedDotDict classes that allow to enforce typing of dictionaries.
- Bug fixes: improved internal logic of rendering data documentation, slack notification, and CLI profile command when datasource argument was not provided.

10.2.31 0.7.5

- Fix missing requirement for pypandoc brought in from markdown support for notes rendering.

10.2.32 0.7.4

- Fix numerous rendering bugs and formatting issues for rendering documentation.
- Add support for pandas extension dtypes in pandas backend of expect_column_values_to_be_of_type and expect_column_values_to_be_in_type_list and fix bug affecting some dtype-based checks.
- Add datetime and boolean column-type detection in BasicDatasetProfiler.
- Improve BasicDatasetProfiler performance by disabling interactive evaluation when output of expectation is not immediately used for determining next expectations in profile.
- Add support for rendering expectation_suite and expectation_level notes from meta in docs.
- Fix minor formatting issue in readthedocs documentation.

10.2.33 0.7.3

- BREAKING: Harmonize expect_column_values_to_be_of_type and expect_column_values_to_be_in_type_list semantics in Pandas with other backends, including support for None type and type_list parameters to support profiling. *These type expectations now rely exclusively on native python or numpy type names.*
- Add configurable support for Custom DataAsset modules to DataContext
- Improve support for setting and inheriting custom data_asset_type names
- Add tooltips with expectations backing data elements to rendered documentation
- Allow better selective disabling of tests (thanks @RoyalITS)
- Fix documentation build errors causing missing code blocks on readthedocs
- Update the parameter naming system in DataContext to reflect data_asset_name *and* expectation_suite_name
- Change scary warning about discarding expectations to be clearer, less scary, and only in log
- Improve profiler support for boolean types, value_counts, and type detection
- Allow user to specify data_assets to profile via CLI
- Support CLI rendering of expectation_suite and EVR-based documentation

10.2.34 0.7.2

- Improved error detection and handling in CLI "add datasource" feature
- Fixes in rendering of profiling results (descriptive renderer of validation results)
- Query Generator of SQLAlchemy datasource adds tables in non-default schemas to the data asset namespace
- Added convenience methods to display HTML renderers of sections in Jupyter notebooks
- Implemented prescriptive rendering of expectations for most expectation types

10.2.35 0.7.1

- Added documentation/tutorials/videos for onboarding and new profiling and documentation features
- Added prescriptive documentation built from expectation suites
- Improved index, layout, and navigation of data context HTML documentation site
- Bug fix: non-Python files were not included in the package
- Improved the rendering logic to gracefully deal with failed expectations
- Improved the basic dataset profiler to be more resilient
- Implement expect_column_values_to_be_of_type, expect_column_values_to_be_in_type_list for SparkDFDataset
- Updated CLI with a new documentation command and improved profile and render commands
- Expectation suites and validation results within a data context are saved in a more readable form (with indentation)
- Improved compatibility between SparkDatasource and InMemoryGenerator
- Optimization for Pandas column type checking
- Optimization for Spark duplicate value expectation (thanks @orenovadia!)
- Default run_id format no longer includes ":" and specifies UTC time
- Other internal improvements and bug fixes

10.2.36 0.7.0

Version 0.7 of Great Expectations is HUGE. It introduces several major new features and a large number of improvements, including breaking API changes.

The core vocabulary of expectations remains consistent. Upgrading to the new version of GE will primarily require changes to code that uses data contexts; existing expectation suites will require only changes to top-level names.

- Major update of Data Contexts. Data Contexts now offer significantly more support for building and maintaining expectation suites and interacting with existing pipeline systems, including providing a namespace for objects.They can handle integrating, registering, and storing validation results, and provide a namespace for data assets, making **batches** first-class citizens in GE. Read more: *DataContexts* or `great_expectations.data_context`
- Major refactor of autoinspect. Autoinspect is now built around a module called "profile" which provides a class-based structure for building expectation suites. There is no longer a default "autoinspect_func" – calling autoinspect requires explicitly passing the desired profiler. See *Profiling*
- New "Compile to Docs" feature produces beautiful documentation from expectations and expectation validation reports, helping keep teams on the same page.
- Name clarifications: we've stopped using the overloaded terms "expectations config" and "config" and instead use "expectation suite" to refer to a collection (or suite!) of expectations that can be used for validating a data asset.
  - Expectation Suites include several top level keys that are useful for organizing content in a data context: data_asset_name, expectation_suite_name, and data_asset_type. When a data_asset is validated, those keys will be placed in the *meta* key of the validation result.
- Major enhancement to the CLI tool including *init*, *render* and more flexibility with *validate*

- Added helper notebooks to make it easy to get started. Each notebook acts as a combination of tutorial and code scaffolding, to help you quickly learn best practices by applying them to your own data.
- Relaxed constraints on expectation parameter values, making it possible to declare many column aggregate expectations in a way that is always "vacuously" true, such as `expect_column_values_to_be_between None` and `None`. This makes it possible to progressively tighten expectations while using them as the basis for profiling results and documentation.
- Enabled caching on dataset objects by default.
- Bugfixes and improvements:
  - New expectations:
    - expect_column_quantile_values_to_be_between
    - expect_column_distinct_values_to_be_in_set
  - Added support for `head` method on all current backends, returning a PandasDataset
  - More implemented expectations for SparkDF Dataset with optimizations
    - expect_column_values_to_be_between
    - expect_column_median_to_be_between
    - expect_column_value_lengths_to_be_between
  - Optimized histogram fetching for SqlalchemyDataset and SparkDFDataset
  - Added cross-platform internal partition method, paving path for improved profiling
  - Fixed bug with outputstrftime not being honored in PandasDataset
  - Fixed series naming for column value counts
  - Standardized naming for expect_column_values_to_be_of_type
  - Standardized and made explicit use of sample normalization in stdev calculation
  - Added from_dataset helper
  - Internal testing improvements
  - Documentation reorganization and improvements
  - Introduce custom exceptions for more detailed error logs

10.2.37 0.6.1

- Re-add testing (and support) for py2
- NOTE: Support for SqlAlchemyDataset and SparkDFDataset is enabled via optional install (e.g. `pip install great_expectations[sqlalchemy]` or `pip install great_expectations[spark]`)

10.2.38 0.6.0

- Add support for SparkDFDataset and caching (HUGE work from @cselig)
- Migrate distributional expectations to new testing framework
- Add support for two new expectations: expect_column_distinct_values_to_contain_set and expect_column_distinct_values_to_equal_set (thanks @RoyalTS)
- FUTURE BREAKING CHANGE: The new cache mechanism for Datasets, when enabled, causes GE to assume that dataset does not change between evaluation of individual expectations. We anticipate this will become the future default behavior.
- BREAKING CHANGE: Drop official support pandas < 0.22

10.2.39 0.5.1

- **Fix** issue where no result_format available for expect_column_values_to_be_null caused error
- Use vectorized computation in pandas (#443, #445; thanks @RoyalTS)

10.2.40 0.5.0

- Restructured class hierarchy to have a more generic DataAsset parent that maintains expectation logic separate from the tabular organization of Dataset expectations
- Added new FileDataAsset and associated expectations (#416 thanks @anhollis)
- Added support for date/datetime type columns in some SQLAlchemy expectations (#413)
- Added support for a multicolumn expectation, expect multicolumn values to be unique (#408)
- **Optimization**: You can now disable *partial_unexpected_counts* by setting the *partial_unexpected_count* value to 0 in the result_format argument, and we do not compute it when it would not be returned. (#431, thanks @eugmandel)
- **Fix**: Correct error in unexpected_percent computations for sqlalchemy when unexpected values exceed limit (#424)
- **Fix**: Pass meta object to expectation result (#415, thanks @jseeman)
- Add support for multicolumn expectations, with *expect_multicolumn_values_to_be_unique* as an example (#406)
- Add dataset class to from_pandas to simplify using custom datasets (#404, thanks @jtilly)
- Add schema support for sqlalchemy data context (#410, thanks @rahulj51)
- Minor documentation, warning, and testing improvements (thanks @zdog).

10.2.41 0.4.5

- Add a new autoinspect API and remove default expectations.
- Improve details for expect_table_columns_to_match_ordered_list (#379, thanks @rlshuhart)
- Linting fixes (thanks @elsander)
- Add support for dataset_class in from_pandas (thanks @jtilly)
- Improve redshift compatibility by correcting faulty isnull operator (thanks @avanderm)
- Adjust partitions to use tail_weight to improve JSON compatibility and support special cases of KL Divergence (thanks @anhollis)
- Enable custom_sql datasets for databases with multiple schemas, by adding a fallback for column reflection (#387, thanks @elsander)
- Remove *IF NOT EXISTS* check for custom sql temporary tables, for Redshift compatibility (#372, thanks @elsander)
- Allow users to pass args/kwargs for engine creation in SqlAlchemyDataContext (#369, thanks @elsander)
- Add support for custom schema in SqlAlchemyDataset (#370, thanks @elsander)
- Use getfullargspec to avoid deprecation warnings.
- Add expect_column_values_to_be_unique to SqlAlchemyDataset
- **Fix** map expectations for categorical columns (thanks @eugmandel)
- Improve internal testing suite (thanks @anhollis and @ccnobbli)
- Consistently use value_set instead of mixing value_set and values_set (thanks @njsmith8)

10.2.42 0.4.4

- Improve CLI help and set CLI return value to the number of unmet expectations
- Add error handling for empty columns to SqlAlchemyDataset, and associated tests
- **Fix** broken support for older pandas versions (#346)
- **Fix** pandas deepcopy issue (#342)

10.2.43 0.4.3

- Improve type lists in expect_column_type_to_be[_in_list] (thanks @smontanaro and @ccnobbli)
- Update cli to use entry_points for conda compatibility, and add version option to cli
- Remove extraneous development dependency to airflow
- Address SQlAlchemy warnings in median computation
- Improve glossary in documentation
- Add 'statistics' section to validation report with overall validation results (thanks @sotte)
- Add support for parameterized expectations
- Improve support for custom expectations with better error messages (thanks @syk0saje)
- Implement expect_column_value_lenghts_to_[be_between|equal] for SQAlchemy (thanks @ccnobbli)

- **Fix** PandasDataset subclasses to inherit child class

10.2.44 0.4.2

- **Fix** bugs in expect_column_values_to_[not]_be_null: computing unexpected value percentages and handling all-null (thanks @ccnobbli)
- Support mysql use of Decimal type (thanks @bouke-nederstigt)
- Add new expectation expect_column_values_to_not_match_regex_list.
  - Change behavior of expect_column_values_to_match_regex_list to use python re.findall in PandasDataset, relaxing matching of individuals expressions to allow matches anywhere in the string.
- **Fix** documentation errors and other small errors (thanks @roblim, @ccnobbli)

10.2.45 0.4.1

- Correct inclusion of new data_context module in source distribution

10.2.46 0.4.0

- Initial implementation of data context API and SqlAlchemyDataset including implementations of the following expectations:
  - expect_column_to_exist
  - expect_table_row_count_to_be
  - expect_table_row_count_to_be_between
  - expect_column_values_to_not_be_null
  - expect_column_values_to_be_null
  - expect_column_values_to_be_in_set
  - expect_column_values_to_be_between
  - expect_column_mean_to_be
  - expect_column_min_to_be
  - expect_column_max_to_be
  - expect_column_sum_to_be
  - expect_column_unique_value_count_to_be_between
  - expect_column_proportion_of_unique_values_to_be_between
- Major refactor of output_format to new result_format parameter. See docs for full details:
  - exception_list and related uses of the term exception have been renamed to unexpected
  - Output formats are explicitly hierarchical now, with BOOLEAN_ONLY < BASIC < SUMMARY < COMPLETE. All *column_aggregate_expectation* expectations now return element count and related information included at the BASIC level or higher.
- New expectation available for parameterized distributions–expect_column_parameterized_distribution_ks_test_p_value_to_be_gr
(what a name! :) – (thanks @ccnobbli)

- ge.from_pandas() utility (thanks @schrockn)
- Pandas operations on a PandasDataset now return another PandasDataset (thanks @dlwhite5)
- expect_column_to_exist now takes a column_index parameter to specify column order (thanks @louispotok)
- Top-level validate option (ge.validate())
- ge.read_json() helper (thanks @rjurney)
- Behind-the-scenes improvements to testing framework to ensure parity across data contexts.
- Documentation improvements, bug-fixes, and internal api improvements

10.2.47 0.3.2

- Include requirements file in source dist to support conda

10.2.48 0.3.1

- **Fix** infinite recursion error when building custom expectations
- Catch dateutil parsing overflow errors

10.2.49 0.2

- Distributional expectations and associated helpers are improved and renamed to be more clear regarding the tests they apply
- Expectation decorators have been refactored significantly to streamline implementing expectations and support custom expectations
- API and examples for custom expectations are available
- New output formats are available for all expectations
- Significant improvements to test suite and compatibility

CHAPTER

ELEVEN

MODULE DOCS

11.1 DataAsset Module

class great_expectations.data_asset.data_asset.**DataAsset**(**args*, ***kwargs*)

autoinspect(*profiler*)
Deprecated: use profile instead.

Use the provided profiler to evaluate this data_asset and assign the resulting expectation suite as its own.

Parameters **profiler** – The profiler to use

Returns tuple(expectation_suite, validation_results)

profile(*profiler*)
Use the provided profiler to evaluate this data_asset and assign the resulting expectation suite as its own.

Parameters **profiler** – The profiler to use

Returns tuple(expectation_suite, validation_results)

edit_expectation_suite()

classmethod expectation(*method_arg_names*)
Manages configuration and running of expectation objects.

Expectation builds and saves a new expectation configuration to the DataAsset object. It is the core decorator used by great expectations to manage expectation configurations.

Parameters **method_arg_names** (*List*) – An ordered list of the arguments used by the method implementing the expectation (typically the result of inspection). Positional arguments are explicitly mapped to keyword arguments when the expectation is run.

Notes

Intermediate decorators that call the core @expectation decorator will most likely need to pass their decorated methods' signature up to the expectation decorator. For example, the MetaPandasDataset column_map_expectation decorator relies on the DataAsset expectation decorator, but will pass through the signature from the implementing method.

@expectation intercepts and takes action based on the following parameters:

- include_config (boolean or None) : If True, then include the generated expectation config as part of the result object. For more detail, see *include_config*.

- catch_exceptions (boolean or None) : If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **result_format (str or None)** [Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*.] For more detail, see *result_format*.
- meta (dict or None): A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

append_expectation(*expectation_config*)
This method is a thin wrapper for ExpectationSuite.append_expectation

find_expectation_indexes(*expectation_type=None*, *column=None*, *expectation_kwargs=None*)
This method is a thin wrapper for ExpectationSuite.find_expectation_indexes

find_expectations(*expectation_type=None*, *column=None*, *expectation_kwargs=None*, *discard_result_format_kwargs=True*, *discard_include_config_kwargs=True*, *discard_catch_exceptions_kwargs=True*)
This method is a thin wrapper for ExpectationSuite.find_expectations()

remove_expectation(*expectation_type=None*, *column=None*, *expectation_kwargs=None*, *remove_multiple_matches=False*, *dry_run=False*)
This method is a thin wrapper for ExpectationSuite.remove()

set_config_value(*key*, *value*)

get_config_value(*key*)

property batch_kwargs

property batch_id

property batch_markers

property batch_parameters

discard_failing_expectations()

get_default_expectation_arguments()
Fetch default expectation arguments for this data_asset

Returns

A dictionary containing all the current default expectation arguments for a data_asset

Ex:

```
{
    "include_config" : True,
    "catch_exceptions" : False,
    "result_format" : 'BASIC'
}
```

See also:

set_default_expectation_arguments

set_default_expectation_argument(*argument*, *value*)
Set a default expectation argument for this data_asset

Parameters

- **argument** (*string*) – The argument to be replaced
- **value** – The New argument to use for replacement

Returns None

See also:

get_default_expectation_arguments

get_expectations_config(*discard_failed_expectations=True*, *discard_result_format_kwargs=True*, *discard_include_config_kwargs=True*, *discard_catch_exceptions_kwargs=True*, *suppress_warnings=False*)

get_expectation_suite(*discard_failed_expectations=True*, *discard_result_format_kwargs=True*, *discard_include_config_kwargs=True*, *discard_catch_exceptions_kwargs=True*, *suppress_warnings=False*)

Returns _expectation_config as a JSON object, and perform some cleaning along the way.

Parameters

- **discard_failed_expectations** (*boolean*) – Only include expectations with success_on_last_run=True in the exported config. Defaults to *True*.
- **discard_result_format_kwargs** (*boolean*) – In returned expectation objects, suppress the *result_format* parameter. Defaults to *True*.
- **discard_include_config_kwargs** (*boolean*) – In returned expectation objects, suppress the *include_config* parameter. Defaults to *True*.
- **discard_catch_exceptions_kwargs** (*boolean*) – In returned expectation objects, suppress the *catch_exceptions* parameter. Defaults to *True*.

Returns An expectation suite.

Note: get_expectation_suite does not affect the underlying expectation suite at all. The returned suite is a copy of _expectation_suite, not the original object.

---

save_expectation_suite(*filepath=None*, *discard_failed_expectations=True*, *discard_result_format_kwargs=True*, *discard_include_config_kwargs=True*, *discard_catch_exceptions_kwargs=True*, *suppress_warnings=False*)

Writes `_expectation_config` to a JSON file.

Writes the DataAsset's expectation config to the specified JSON `filepath`. Failing expectations can be excluded from the JSON expectations config with `discard_failed_expectations`. The kwarg key-value pairs *result_format*, *include_config*, and *catch_exceptions* are optionally excluded from the JSON expectations config.

Parameters

- **filepath** (*string*) – The location and name to write the JSON config file to.
- **discard_failed_expectations** (*boolean*) – If True, excludes expectations that do not return `success = True`. If False, all expectations are written to the JSON config file.
- **discard_result_format_kwargs** (*boolean*) – If True, the *result_format* attribute for each expectation is not written to the JSON config file.
- **discard_include_config_kwargs** (*boolean*) – If True, the *include_config* attribute for each expectation is not written to the JSON config file.

- **discard_catch_exceptions_kwargs** (*boolean*) – If True, the *catch_exceptions* attribute for each expectation is not written to the JSON config file.
- **suppress_warnings** (*boolean*) – It True, all warnings raised by Great Expectations, as a result of dropped expectations, are suppressed.

validate(*expectation_suite=None*, *run_id=None*, *data_context=None*, *evaluation_parameters=None*, *catch_exceptions=True*, *result_format=None*, *only_return_failures=False*)

Generates a JSON-formatted report describing the outcome of all expectations.

Use the default expectation_suite=None to validate the expectations config associated with the DataAsset.

Parameters

- **expectation_suite** (*json or None*) – If None, uses the expectations config generated with the DataAsset during the current session. If a JSON file, validates those expectations.
- **run_id** (*str*) – A string used to identify this validation result as part of a collection of validations. See DataContext for more information.
- **data_context** (DataContext) – A datacontext object to use as part of validation for binding evaluation parameters and registering validation results.
- **evaluation_parameters** (*dict or None*) – If None, uses the evaluation_paramters from the expectation_suite provided or as part of the data_asset. If a dict, uses the evaluation parameters in the dictionary.
- **catch_exceptions** (*boolean*) – If True, exceptions raised by tests will not end validation and will be described in the returned report.
- **result_format** (*string or None*) – If None, uses the default value ('BASIC' or as specified). If string, the returned expectation output follows the specified format ('BOOLEAN_ONLY','BASIC', etc.).
- **only_return_failures** (*boolean*) – If True, expectation results are only returned when `success = False`

Returns

A JSON-formatted dictionary containing a list of the validation results. An example of the returned format:

```
{
  "results": [
    {
      "unexpected_list": [unexpected_value_1, unexpected_value_2],
      "expectation_type": "expect_*",
      "kwargs": {
        "column": "Column_Name",
        "output_format": "SUMMARY"
      },
      "success": true,
      "raised_exception: false.
      "exception_traceback": null
    },
    {
      ... (Second expectation results)
    },
    ... (More expectations results)
```

(continues on next page)

(continued from previous page)

```
  ],
  "success": true,
  "statistics": {
    "evaluated_expectations": n,
    "successful_expectations": m,
    "unsuccessful_expectations": n - m,
    "success_percent": m / n
  }
}
```

Notes

If the configuration object was built with a different version of great expectations then the current environment. If no version was found in the configuration file.

Raises **AttributeError - if 'catch_exceptions'=None and an expectation throws an AttributeError** –

get_evaluation_parameter(*parameter_name*, *default_value=None*)
Get an evaluation parameter value that has been stored in meta.

Parameters

- **parameter_name** (*string*) – The name of the parameter to store.
- **default_value** (*any*) – The default value to be returned if the parameter is not found.

Returns The current value of the evaluation parameter.

set_evaluation_parameter(*parameter_name*, *parameter_value*)
Provide a value to be stored in the data_asset evaluation_parameters object and used to evaluate parameterized expectations.

Parameters

- **parameter_name** (*string*) – The name of the kwarg to be replaced at evaluation time
- **parameter_value** (*any*) – The value to be used

add_citation(*comment*, *batch_kwargs=None*, *batch_markers=None*, *batch_parameters=None*, *citation_date=None*)

property expectation_suite_name
Gets the current expectation_suite name of this data_asset as stored in the expectations configuration.

test_expectation_function(*function*, **args*, ***kwargs*)
Test a generic expectation function

Parameters

- **function** (*func*) – The function to be tested. (Must be a valid expectation function.)
- ***args** – Positional arguments to be passed the the function
- ****kwargs** – Keyword arguments to be passed the the function

Returns A JSON-serializable expectation result object.

Notes

This function is a thin layer to allow quick testing of new expectation functions, without having to define custom classes, etc. To use developed expectations from the command-line tool, you will still need to define custom classes, etc.

Check out *Building Custom Expectations* for more information.

class great_expectations.data_asset.data_asset.**ValidationStatistics**(*evaluated_expectations*, *successful_expectations*, *unsuccessful_expectations*, *success_percent*, *success*)

Bases: tuple

property evaluated_expectations
Alias for field number 0

property success
Alias for field number 4

property success_percent
Alias for field number 3

property successful_expectations
Alias for field number 1

property unsuccessful_expectations
Alias for field number 2

11.1.1 FileDataAsset

class great_expectations.data_asset.file_data_asset.**MetaFileDataAsset**(**args*, ***kwargs*)

Bases: *great_expectations.data_asset.data_asset.DataAsset*

MetaFileDataset is a thin layer above FileDataset. This two-layer inheritance is required to make @classmethod decorators work. Practically speaking, that means that MetaFileDataset implements expectation decorators, like *file_lines_map_expectation* and FileDataset implements the expectation methods themselves.

classmethod file_lines_map_expectation(*func*)
Constructs an expectation using file lines map semantics. The file_lines_map_expectations decorator handles boilerplate issues surrounding the common pattern of evaluating truthiness of some condition on an line by line basis in a file.

Parameters **func** (*function*) – The function implementing an expectation that will be applied line by line across a file. The function should take a file and return information about how many lines met expectations.

Notes

Users can specify skip value k that will cause the expectation function to disregard the first k lines of the file.

file_lines_map_expectation will add a kwarg _lines to the called function with the nonnull lines to process.

null_lines_regex defines a regex used to skip lines, but can be overridden

See also:

`expect_file_line_regex_match_count_to_be_between` for an example of a file_lines_map_expectation

class great_expectations.data_asset.file_data_asset.**FileDataAsset**(*file_path=None*, **args*, ***kwargs*)

Bases: *great_expectations.data_asset.file_data_asset.MetaFileDataAsset*

FileDataset instantiates the great_expectations Expectations API as a subclass of a python file object. For the full API reference, please see `DataAsset`

expect_file_line_regex_match_count_to_be_between(*regex*, *expected_min_count=0*, *expected_max_count=None*, *skip=None*, *mostly=None*, *null_lines_regex='^\\s*$'*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*, *_lines=None*)

Expect the number of times a regular expression appears on each line of a file to be between a maximum and minimum value.

Parameters

- **regex** – A string that can be compiled as valid regular expression to match
- **expected_min_count** (*None or nonnegative integer*) – Specifies the minimum number of times regex is expected to appear on each line of the file
- **expected_max_count** (*None or nonnegative integer*) – Specifies the maximum number of times regex is expected to appear on each line of the file

Keyword Arguments

- **skip** (*None or nonnegative integer*) – Integer specifying the first lines in the file the method should skip before assessing expectations
- **mostly** (*None or number between 0 and 1*) – Specifies an acceptable error for expectations. If the percentage of unexpected lines is less than mostly, the method still returns true even if all lines don't match the expectation criteria.
- **null_lines_regex** (*valid regular expression or None*) – If not none, a regex to skip lines as null. Defaults to empty or whitespace-only lines.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.

- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.
- **_lines** (*list*) – The lines over which to operate (provided by the file_lines_map_expectation decorator)

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_file_line_regex_match_count_to_equal(*regex*, *expected_count=0*, *skip=None*, *mostly=None*, *nonnull_lines_regex='^\\s*$'*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*, *_lines=None*)

Expect the number of times a regular expression appears on each line of a file to be between a maximum and minimum value.

Parameters

- **regex** – A string that can be compiled as valid regular expression to match
- **expected_count** (*None or nonnegative integer*) – Specifies the number of times regex is expected to appear on each line of the file

Keyword Arguments

- **skip** (*None or nonnegative integer*) – Integer specifying the first lines in the file the method should skip before assessing expectations
- **mostly** (*None or number between 0 and 1*) – Specifies an acceptable error for expectations. If the percentage of unexpected lines is less than mostly, the method still returns true even if all lines don't match the expectation criteria.
- **null_lines_regex** (*valid regular expression or None*) – If not none, a regex to skip lines as null. Defaults to empty or whitespace-only lines.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.
- **_lines** (*list*) – The lines over which to operate (provided by the file_lines_map_expectation decorator)

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_file_hash_to_equal (*value*, *hash_alg='md5'*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect computed file hash to equal some given value.

Parameters **value** – A string to compare with the computed hash value

Keyword Arguments

- **hash_alg** (*string*) – Indicates the hash algorithm to use
- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_file_size_to_be_between (*minsize=0*, *maxsize=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect file size to be between a user specified maxsize and minsize.

Parameters

- **minsize** (*integer*) – minimum expected file size
- **maxsize** (*integer*) – maximum expected file size

Keyword Arguments

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_file_to_exist (*filepath=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Checks to see if a file specified by the user actually exists

Parameters **filepath** (*str or None*) – The filepath to evalutate. If none, will check the currently-configured path object of this FileDataAsset.

Keyword Arguments

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_file_to_have_valid_table_header(*regex*, *skip=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Checks to see if a file has a line with unique delimited values, such a line may be used as a table header.

Keyword Arguments

- **skip** (*nonnegative integer*) – Integer specifying the first lines in the file the method should skip before assessing expectations
- **regex** (*string*) – A string that can be compiled as valid regular expression. Used to specify the elements of the table header (the column headers)
- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_file_to_be_valid_json(*schema=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Parameters

- **schema** – string optional JSON schema file on which JSON data file is validated against
- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.

- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification.

For more detail, see *meta*.

Returns A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

11.1.2 util

great_expectations.data_asset.util.**parse_result_format**(*result_format*)
This is a simple helper utility that can be used to parse a string result_format into the dict format used internally by great_expectations. It is not necessary but allows shorthand for result_format in cases where there is no need to specify a custom partial_unexpected_count.

class great_expectations.data_asset.util.**DocInherit**(*mthd*)
Bases: object

great_expectations.data_asset.util.**recursively_convert_to_json_serializable**(*test_obj*)
Helper function to convert a dict object to one that is serializable

Parameters **test_obj** – an object to attempt to convert a corresponding json-serializable object

Returns (dict) A converted test_object

Warning: test_obj may also be converted in place.

11.2 Dataset Module

11.2.1 Dataset

class great_expectations.dataset.dataset.**MetaDataset**(**args*, ***kwargs*)
Bases: *great_expectations.data_asset.data_asset.DataAsset*

Holds expectation decorators.

classmethod column_map_expectation(*func*)
Constructs an expectation using column-map semantics.

The column_map_expectation decorator handles boilerplate issues surrounding the common pattern of evaluating truthiness of some condition on a per-row basis.

Parameters **func** (*function*) – The function implementing a row-wise expectation. The function should take a column of data and return an equally-long column of boolean values corresponding to the truthiness of the underlying expectation.

Notes

column_map_expectation intercepts and takes action based on the following parameters: mostly (None or a float between 0 and 1): Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

column_map_expectation *excludes null values* from being passed to the function

Depending on the *result_format* selected, column_map_expectation can additional data to a return object, including *element_count*, *nonnull_values*, *nonnull_count*, *success_count*, *unexpected_list*, and *unexpected_index_list*. See `_format_map_output`

See also:

expect_column_values_to_be_in_set for an example of a column_map_expectation

classmethod column_aggregate_expectation(*func*)
Constructs an expectation using column-aggregate semantics.

The column_aggregate_expectation decorator handles boilerplate issues surrounding the common pattern of evaluating truthiness of some condition on an aggregated-column basis.

Parameters **func** (*function*) – The function implementing an expectation using an aggregate property of a column. The function should take a column of data and return the aggregate value it computes.

Notes

column_aggregate_expectation *excludes null values* from being passed to the function

See also:

expect_column_mean_to_be_between for an example of a column_aggregate_expectation

class great_expectations.dataset.dataset.**Dataset**(**args*, ***kwargs*)
Bases: *great_expectations.dataset.dataset.MetaDataset*

hashable_getters = ['get_column_min', 'get_column_max', 'get_column_mean', 'get_column

classmethod from_dataset(*dataset=None*)
This base implementation naively passes arguments on to the real constructor, which is suitable really when a constructor knows to take its own type. In general, this should be overridden

get_row_count()
Returns: int, table row count

get_column_count()
Returns: int, table column count

get_table_columns()
Returns: List[str], list of column names

get_column_nonnull_count(*column*)
Returns: int

get_column_mean(*column*)
Returns: float

get_column_value_counts(*column*, *sort='value'*, *collate=None*)
Get a series containing the frequency counts of unique values from the named column.

Parameters

- **column** – the column for which to obtain value_counts
- **sort** (*string*) – must be one of "value", "count", or "none". - if "value" then values in the resulting partition object will be sorted lexigraphically - if "count" then values will be sorted according to descending count (frequency) - if "none" then values will not be sorted
- **collate** (*string*) – the collate (sort) method to be used on supported backends (SqlAlchemy only)

Returns pd.Series of value counts for a column, sorted according to the value requested in sort

get_column_sum(*column*)
Returns: float

get_column_max(*column*, *parse_strings_as_datetimes=False*)
Returns: any

get_column_min(*column*, *parse_strings_as_datetimes=False*)
Returns: any

get_column_unique_count(*column*)
Returns: int

get_column_modes(*column*)
Returns: List[any], list of modes (ties OK)

get_column_median(*column*)
Returns: any

get_column_quantiles(*column*, *quantiles*, *allow_relative_error=False*)
Get the values in column closest to the requested quantiles :param column: name of column :type column: string :param quantiles: the quantiles to return. quantiles *must* be a tuple to ensure caching is possible :type quantiles: tuple of float

Returns the nearest values in the dataset to those quantiles

Return type List[any]

get_column_stdev(*column*)
Returns: float

get_column_partition(*column*, *bins='uniform'*, *n_bins=10*, *allow_relative_error=False*)
Get a partition of the range of values in the specified column.

Parameters

- **column** – the name of the column
- **bins** – 'uniform' for evenly spaced bins or 'quantile' for bins spaced according to quantiles
- **n_bins** – the number of bins to produce
- **allow_relative_error** – passed to get_column_quantiles, set to False for only precise values, True to allow approximate values on systems with only binary choice (e.g. Redshift), and to a value between zero and one for systems that allow specification of relative error (e.g. SparkDFDataset).

Returns A list of bins

get_column_hist(*column*, *bins*)
Get a histogram of column values :param column: the column for which to generate the histogram :param bins: the bins to slice the histogram. bins *must* be a tuple to ensure caching is possible :type bins: tuple

Returns: List[int], a list of counts corresponding to bins

get_column_count_in_range(*column*, *min_val=None*, *max_val=None*, *strict_min=False*, *strict_max=True*)

Returns: int

test_column_map_expectation_function(*function*, **args*, ***kwargs*)

Test a column map expectation function

Parameters

- **function** (*func*) – The function to be tested. (Must be a valid column_map_expectation function.)
- ***args** – Positional arguments to be passed the the function
- ****kwargs** – Keyword arguments to be passed the the function

Returns A JSON-serializable expectation result object.

Notes

This function is a thin layer to allow quick testing of new expectation functions, without having to define custom classes, etc. To use developed expectations from the command-line tool, you'll still need to define custom classes, etc.

Check out *Building Custom Expectations* for more information.

test_column_aggregate_expectation_function(*function*, **args*, ***kwargs*)

Test a column aggregate expectation function

Parameters

- **function** (*func*) – The function to be tested. (Must be a valid column_aggregate_expectation function.)
- ***args** – Positional arguments to be passed the the function
- ****kwargs** – Keyword arguments to be passed the the function

Returns A JSON-serializable expectation result object.

Notes

This function is a thin layer to allow quick testing of new expectation functions, without having to define custom classes, etc. To use developed expectations from the command-line tool, you'll still need to define custom classes, etc.

Check out *Building Custom Expectations* for more information.

expect_column_to_exist(*column*, *column_index=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the specified column to exist.

expect_column_to_exist is a *expectation*, not a `column_map_expectation` or `column_aggregate_expectation`.

Parameters **column** (*str*) – The column name.

Other Parameters

- **column_index** (*int or None*) – If not None, checks the order of the columns. The expectation will fail if the column is not in location column_index (zero-indexed).

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_table_columns_to_match_ordered_list (*column_list*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the columns to exactly match a specified list.

expect_table_columns_to_match_ordered_list is a *expectation*, not a column_map_expectation or column_aggregate_expectation.

Parameters **column_list** (*list of str*) – The column names, in the correct order.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_table_column_count_to_be_between (*min_value=None*, *max_value=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the number of columns to be between two values.

expect_table_column_count_to_be_between is a *expectation*, not a column_map_expectation or column_aggregate_expectation.

Keyword Arguments

- **min_value** (*int or None*) – The minimum number of columns, inclusive.
- **max_value** (*int or None*) – The maximum number of columns, inclusive.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

- min_value and max_value are both inclusive.
- If min_value is None, then max_value is treated as an upper bound, and the number of acceptable columns has no minimum.
- If max_value is None, then min_value is treated as a lower bound, and the number of acceptable columns has no maximum.

See also:

expect_table_column_count_to_equal

expect_table_column_count_to_equal(*value*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the number of columns to equal a value.

expect_table_column_count_to_equal is a `expectation`, not a `column_map_expectation` or `column_aggregate_expectation`.

Parameters **value** (*int*) – The expected number of columns.

Other Parameters

- **result_format** (*string or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_table_column_count_to_be_between

expect_table_row_count_to_be_between(*min_value=None*, *max_value=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the number of rows to be between two values.

expect_table_row_count_to_be_between is a *expectation*, not a `column_map_expectation` or `column_aggregate_expectation`.

Keyword Arguments

- **min_value** (*int or None*) – The minimum number of rows, inclusive.
- **max_value** (*int or None*) – The maximum number of rows, inclusive.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

- min_value and max_value are both inclusive.
- If min_value is None, then max_value is treated as an upper bound, and the number of acceptable rows has no minimum.
- If max_value is None, then min_value is treated as a lower bound, and the number of acceptable rows has no maximum.

See also:

expect_table_row_count_to_equal

expect_table_row_count_to_equal(*value*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the number of rows to equal a value.

expect_table_row_count_to_equal is a *expectation*, not a `column_map_expectation` or `column_aggregate_expectation`.

Parameters **value** (*int*) – The expected number of rows.

Other Parameters

- **result_format** (*string or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_table_row_count_to_be_between

expect_column_values_to_be_unique(*column*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect each column value to be unique.

This expectation detects duplicates. All duplicated values are counted as exceptions.

For example, *[1, 2, 3, 3, 3]* will return *[3, 3, 3]* in *result.exceptions_list*, with *unexpected_percent = 60.0*.

expect_column_values_to_be_unique is a `column_map_expectation`.

Parameters **column** (`str`) – The column name.

Keyword Arguments **mostly** (`None or a float between 0 and 1`) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_column_values_to_not_be_null(*column*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column values to not be null.

To be counted as an exception, values must be explicitly null or missing, such as a NULL in PostgreSQL or an np.NaN in pandas. Empty strings don't count as null unless they have been coerced to a null type.

expect_column_values_to_not_be_null is a *column_map_expectation*.

Parameters **column** (*str*) – The column name.

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_be_null

expect_column_values_to_be_null(*column*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column values to be null.

expect_column_values_to_be_null is a *column_map_expectation*.

Parameters **column** (*str*) – The column name.

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_not_be_null

expect_column_values_to_be_of_type(*column*, *type_*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect a column to contain values of a specified data type.

expect_column_values_to_be_of_type is a *column_aggregate_expectation* for typed-column backends, and also for PandasDataset where the column dtype and provided **type_** are unambiguous constraints (any dtype except 'object' or dtype of 'object' with **type_** specified as 'object').

For PandasDataset columns with dtype of 'object' expect_column_values_to_be_of_type is a *column_map_expectation* and will independently check each row's type.

Parameters

- **column** (*str*) – The column name.
- **type_** (*str*) – A string representing the data type that each column should have as entries. Valid types are defined by the current backend implementation and are dynamically loaded. For example, valid types for PandasDataset include any numpy dtype values (such as 'int64') or native python types (such as 'int'), whereas valid types for a SqlAlchemyDataset include types named by the current driver such as 'INTEGER' in most SQL dialects and 'TEXT' in dialects such as postgresql. Valid types for SparkDFDataset include 'StringType', 'BooleanType' and other pyspark-defined type names.

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_be_in_type_list

expect_column_values_to_be_in_type_list(*column*, *type_list*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect a column to contain values from a specified type list.

expect_column_values_to_be_in_type_list is a *column_aggregate_expectation* for typed-column backends, and also for PandasDataset where the column dtype provides an unambiguous constraints (any dtype except 'object'). For PandasDataset columns with dtype of 'object' expect_column_values_to_be_of_type is a *column_map_expectation* and will independently check each row's type.

Parameters

- **column** (*str*) – The column name.
- **type_list** (*str*) – A list of strings representing the data type that each column should have as entries. Valid types are defined by the current backend implementation and are dynamically loaded. For example, valid types for PandasDataset include any numpy dtype values (such as 'int64') or native python types (such as 'int'), whereas valid types for a SqlAlchemyDataset include types named by the current driver such as 'INTEGER' in most SQL dialects and 'TEXT' in dialects such as postgresql. Valid types for SparkDFDataset include 'StringType', 'BooleanType' and other pyspark-defined type names.

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_be_of_type

expect_column_values_to_be_in_set (*column*, *value_set*, *mostly=None*, *parse_strings_as_datetimes=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect each column value to be in a given set.

For example:

```
# my_df.my_col = [1,2,2,3,3,3]
>>> my_df.expect_column_values_to_be_in_set(
    "my_col",
    [2,3]
)
{
  "success": false
```

(continues on next page)

(continued from previous page)

```
  "result": {
    "unexpected_count": 1
    "unexpected_percent": 16.6666666666666666,
    "unexpected_percent_nonmissing": 16.6666666666666666,
    "partial_unexpected_list": [
      1
    ],
  },
}
```

expect_column_values_to_be_in_set is a *column_map_expectation*.

Parameters

- **column** (*str*) – The column name.
- **value_set** (*set-like*) – A set of objects used for comparison.

Keyword Arguments

- **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.
- **parse_strings_as_datetimes** (*boolean or None*) – If True values provided in value_set will be parsed as datetimes before making comparisons.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_not_be_in_set

expect_column_values_to_not_be_in_set(*column*, *value_set*, *mostly=None*, *parse_strings_as_datetimes=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column entries to not be in the set.

For example:

```
# my_df.my_col = [1,2,2,3,3,3]
>>> my_df.expect_column_values_to_not_be_in_set(
    "my_col",
```

(continues on next page)

(continued from previous page)

```
    [1,2]
)
{
  "success": false
  "result": {
    "unexpected_count": 3
    "unexpected_percent": 50.0,
    "unexpected_percent_nonmissing": 50.0,
    "partial_unexpected_list": [
      1, 2, 2
    ],
  },
}
```

expect_column_values_to_not_be_in_set is a *column_map_expectation*.

Parameters

- **column** (*str*) – The column name.
- **value_set** (*set-like*) – A set of objects used for comparison.

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_be_in_set

expect_column_values_to_be_between(*column*, *min_value=None*, *max_value=None*, *strict_min=False*, *strict_max=False*, *allow_cross_type_comparisons=None*, *parse_strings_as_datetimes=False*, *output_strftime_format=None*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column entries to be between a minimum value and a maximum value (inclusive).

expect_column_values_to_be_between is a *column_map_expectation*.

Parameters

- **column** (*str*) – The column name.
- **min_value** (*comparable type or None*) – The minimum value for a column entry.
- **max_value** (*comparable type or None*) – The maximum value for a column entry.

Keyword Arguments

- **strict_min** (*boolean*) – If True, values must be strictly larger than min_value, default=False
- **strict_max** (*boolean*) – If True, values must be strictly smaller than max_value, default=False allow_cross_type_comparisons (boolean or None) : If True, allow comparisons between types (e.g. integer and string). Otherwise, attempting such comparisons will raise an exception.
- **parse_strings_as_datetimes** (*boolean or None*) – If True, parse min_value, max_value, and all non-null column values to datetimes before making comparisons.
- **output_strftime_format** (*str or None*) – A valid strfime format for datetime output. Only used if parse_strings_as_datetimes=True.
- **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

- min_value and max_value are both inclusive unless strict_min or strict_max are set to True.
- If min_value is None, then max_value is treated as an upper bound, and there is no minimum value checked.
- If max_value is None, then min_value is treated as a lower bound, and there is no maximum value checked.

See also:

expect_column_value_lengths_to_be_between

expect_column_values_to_be_increasing(*column*, *strictly=None*, *parse_strings_as_datetimes=False*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column values to be increasing.

By default, this expectation only works for numeric or datetime data. When *parse_strings_as_datetimes=True*, it can also parse strings to datetimes.

If *strictly=True*, then this expectation is only satisfied if each consecutive value is strictly increasing–equal values are treated as failures.

expect_column_values_to_be_increasing is a *column_map_expectation*.

Parameters **column** (*str*) – The column name.

Keyword Arguments

- **strictly** (*Boolean or None*) – If True, values must be strictly greater than previous values
- **parse_strings_as_datetimes** (*boolean or None*) – If True, all non-null column values to datetimes before making comparisons
- **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_be_decreasing

expect_column_values_to_be_decreasing(*column*, *strictly=None*, *parse_strings_as_datetimes=False*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column values to be decreasing.

By default, this expectation only works for numeric or datetime data. When *parse_strings_as_datetimes=True*, it can also parse strings to datetimes.

If *strictly=True*, then this expectation is only satisfied if each consecutive value is strictly decreasing–equal values are treated as failures.

expect_column_values_to_be_decreasing is a *column_map_expectation*.

Parameters **column** (*str*) – The column name.

Keyword Arguments

- **strictly** (*Boolean or None*) – If True, values must be strictly greater than previous values
- **parse_strings_as_datetimes** (*boolean or None*) – If True, all non-null column values to datetimes before making comparisons
- **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_be_increasing

expect_column_value_lengths_to_be_between(*column*, *min_value=None*, *max_value=None*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column entries to be strings with length between a minimum value and a maximum value (inclusive).

This expectation only works for string-type values. Invoking it on ints or floats will raise a TypeError.

expect_column_value_lengths_to_be_between is a *column_map_expectation*.

Parameters **column** (*str*) – The column name.

Keyword Arguments

- **min_value** (*int or None*) – The minimum value for a column entry length.
- **max_value** (*int or None*) – The maximum value for a column entry length.
- **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

- min_value and max_value are both inclusive.
- If min_value is None, then max_value is treated as an upper bound, and the number of acceptable rows has no minimum.
- If max_value is None, then min_value is treated as a lower bound, and the number of acceptable rows has no maximum.

See also:

expect_column_value_lengths_to_equal

expect_column_value_lengths_to_equal(*column*, *value*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column entries to be strings with length equal to the provided value.

This expectation only works for string-type values. Invoking it on ints or floats will raise a TypeError.

expect_column_values_to_be_between is a *column_map_expectation*.

Parameters

- **column** (*str*) – The column name.
- **value** (*int or None*) – The expected value for a column entry length.

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.

- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_value_lengths_to_be_between

expect_column_values_to_match_regex(*column*, *regex*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column entries to be strings that match a given regular expression. Valid matches can be found anywhere in the string, for example "[at]+" will identify the following strings as expected: "cat", "hat", "aa", "a", and "t", and the following strings as unexpected: "fish", "dog".

expect_column_values_to_match_regex is a *column_map_expectation*.

Parameters

- **column** (*str*) – The column name.
- **regex** (*str*) – The regular expression the column entries should match.

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_not_match_regex

expect_column_values_to_match_regex_list

expect_column_values_to_not_match_regex(*column*, *regex*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column entries to be strings that do NOT match a given regular expression. The regex must not match any portion of the provided string. For example, "[at]+" would identify the following strings as expected: "fish", "dog", and the following as unexpected: "cat", "hat".

expect_column_values_to_not_match_regex is a *column_map_expectation*.

Parameters

- **column** (*str*) – The column name.
- **regex** (*str*) – The regular expression the column entries should NOT match.

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_match_regex

expect_column_values_to_match_regex_list

expect_column_values_to_match_regex_list(*column*, *regex_list*, *match_on='any'*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the column entries to be strings that can be matched to either any of or all of a list of regular expressions. Matches can be anywhere in the string.

expect_column_values_to_match_regex_list is a *column_map_expectation*.

Parameters

- **column** (*str*) – The column name.
- **regex_list** (*list*) – The list of regular expressions which the column entries should match

Keyword Arguments

- **match_on=** (*string*) – "any" or "all". Use "any" if the value should match at least one regular expression in the list. Use "all" if it should match each regular expression in the list.
- **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_match_regex

expect_column_values_to_not_match_regex

expect_column_values_to_not_match_regex_list(*column*, *regex_list*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the column entries to be strings that do not match any of a list of regular expressions. Matches can be anywhere in the string.

expect_column_values_to_not_match_regex_list is a *column_map_expectation*.

Parameters

- **column** (*str*) – The column name.
- **regex_list** (*list*) – The list of regular expressions which the column entries should not match

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_match_regex_list

expect_column_values_to_match_strftime_format(*column*, *strftime_format*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column entries to be strings representing a date or time with a given format.

expect_column_values_to_match_strftime_format is a *column_map_expectation*.

Parameters

- **column** (*str*) – The column name.
- **strftime_format** (*str*) – A strftime format string to use for matching

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_column_values_to_be_dateutil_parseable(*column*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column entries to be parsable using dateutil.

expect_column_values_to_be_dateutil_parseable is a *column_map_expectation*.

Parameters **column** (*str*) – The column name.

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.

- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_column_values_to_be_json_parseable(*column*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column entries to be data written in JavaScript Object Notation.

expect_column_values_to_be_json_parseable is a `column_map_expectation`.

Parameters **column** (*str*) – The column name.

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

`expect_column_values_to_match_json_schema`

expect_column_values_to_match_json_schema(*column*, *json_schema*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column entries to be JSON objects matching a given JSON schema.

expect_column_values_to_match_json_schema is a `column_map_expectation`.

Parameters **column** (*str*) – The column name.

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

`expect_column_values_to_be_json_parseable`

The JSON-schema docs.

expect_column_parameterized_distribution_ks_test_p_value_to_be_greater_than(*column*, *distribution*, *p_value=0.05*, *params=None*, *result_format=N*, *include_config=T*, *catch_exceptio*, *meta=None*)

Expect the column values to be distributed similarly to a scipy distribution. This expectation compares the provided column to the specified continuous distribution with a parametric Kolmogorov-Smirnov test. The K-S test compares the provided column to the cumulative density function (CDF) of the specified scipy distribution. If you don't know the desired distribution shape parameters, use the *ge.dataset.util.infer_distribution_parameters()* utility function to estimate them.

It returns 'success'=True if the p-value from the K-S test is greater than or equal to the provided p-value.

`expect_column_parameterized_distribution_ks_test_p_value_to_be_greater_than` is a `column_aggregate_expectation`.

Parameters

- **column** (*str*) – The column name.
- **distribution** (*str*) – The scipy distribution name. See: https://docs.scipy.org/doc/scipy/reference/stats.html Currently supported distributions are listed in the Notes section below.
- **p_value** (*float*) – The threshold p-value for a passing test. Default is 0.05.
- **params** (*dict or list*) – A dictionary or positional list of shape parameters that describe the distribution you want to test the data against. Include key values specific to

the distribution from the appropriate scipy distribution CDF function. 'loc' and 'scale' are used as translational parameters. See https://docs.scipy.org/doc/scipy/reference/stats.html#continuous-distributions

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "details":
        "expected_params" (dict): The specified or inferred parameters of the␣
↪distribution to test                          against
        "ks_results" (dict): The raw result of stats.kstest()
}
```

- The Kolmogorov-Smirnov test's null hypothesis is that the column is similar to the provided distribution.
- Supported scipy distributions:
  - norm
  - beta
  - gamma
  - uniform
  - chi2
  - expon

expect_column_distinct_values_to_be_in_set(*column*, *value_set*, *parse_strings_as_datetimes=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the set of distinct column values to be contained by a given set.

The success value for this expectation will match that of expect_column_values_to_be_in_set. However, expect_column_distinct_values_to_be_in_set is a `column_aggregate_expectation`.

For example:

```
# my_df.my_col = [1,2,2,3,3,3]
>>> my_df.expect_column_distinct_values_to_be_in_set(
    "my_col",
    [2, 3, 4]
)
{
  "success": false
  "result": {
    "observed_value": [1,2,3],
    "details": {
      "value_counts": [
        {
          "value": 1,
          "count": 1
        },
        {
          "value": 2,
          "count": 1
        },
        {
          "value": 3,
          "count": 1
        }
      ]
    }
  }
}
```

Parameters

- **column** (*str*) – The column name.
- **value_set** (*set-like*) – A set of objects used for comparison.

Keyword Arguments **parse_strings_as_datetimes** (*boolean or None*) – If True values provided in value_set will be parsed as datetimes before making comparisons.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_distinct_values_to_contain_set

expect_column_distinct_values_to_equal_set(*column*, *value_set*, *parse_strings_as_datetimes=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the set of distinct column values to equal a given set.

In contrast to expect_column_distinct_values_to_contain_set() this ensures not only that a certain set of values are present in the column but that these *and only these* values are present.

expect_column_distinct_values_to_equal_set is a *column_aggregate_expectation*.

For example:

```
# my_df.my_col = [1,2,2,3,3,3]
>>> my_df.expect_column_distinct_values_to_equal_set(
    "my_col",
    [2,3]
)
{
  "success": false
  "result": {
    "observed_value": [1,2,3]
  },
}
```

Parameters

- **column** (*str*) – The column name.
- **value_set** (*set-like*) – A set of objects used for comparison.

Keyword Arguments **parse_strings_as_datetimes** (*boolean or None*) – If True values provided in value_set will be parsed as datetimes before making comparisons.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_distinct_values_to_contain_set

expect_column_distinct_values_to_contain_set(*column*, *value_set*, *parse_strings_as_datetimes=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the set of distinct column values to contain a given set.

In contrast to expect_column_values_to_be_in_set() this ensures not that all column values are members of the given set but that values from the set *must* be present in the column.

expect_column_distinct_values_to_contain_set is a *column_aggregate_expectation*.

For example:

```
# my_df.my_col = [1,2,2,3,3,3]
>>> my_df.expect_column_distinct_values_to_contain_set(
    "my_col",
    [2,3]
)
{
"success": true
"result": {
    "observed_value": [1,2,3]
},
}
```

Parameters

- **column** (*str*) – The column name.
- **value_set** (*set-like*) – A set of objects used for comparison.

Keyword Arguments **parse_strings_as_datetimes** (*boolean or None*) – If True values provided in value_set will be parsed as datetimes before making comparisons.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_distinct_values_to_equal_set

expect_column_mean_to_be_between(*column*, *min_value=None*, *max_value=None*, *strict_min=False*, *strict_max=False*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the column mean to be between a minimum value and a maximum value (inclusive).

expect_column_mean_to_be_between is a *column_aggregate_expectation*.

Parameters

- **column** (*str*) – The column name.
- **min_value** (*float or None*) – The minimum value for the column mean.
- **max_value** (*float or None*) – The maximum value for the column mean.
- **strict_min** (*boolean*) – If True, the column mean must be strictly larger than min_value, default=False
- **strict_max** (*boolean*) – If True, the column mean must be strictly smaller than max_value, default=False

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "observed_value": (float) The true mean for the column
}
```

- min_value and max_value are both inclusive unless strict_min or strict_max are set to True.
- If min_value is None, then max_value is treated as an upper bound.
- If max_value is None, then min_value is treated as a lower bound.

See also:

expect_column_median_to_be_between

expect_column_stdev_to_be_between

expect_column_median_to_be_between(*column*, *min_value=None*, *max_value=None*, *strict_min=False*, *strict_max=False*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the column median to be between a minimum value and a maximum value.

expect_column_median_to_be_between is a *column_aggregate_expectation*.

Parameters

- **column** (*str*) – The column name.
- **min_value** (*int or None*) – The minimum value for the column median.
- **max_value** (*int or None*) – The maximum value for the column median.
- **strict_min** (*boolean*) – If True, the column median must be strictly larger than min_value, default=False
- **strict_max** (*boolean*) – If True, the column median must be strictly smaller than max_value, default=False

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "observed_value": (float) The true median for the column
}
```

- min_value and max_value are both inclusive unless strict_min or strict_max are set to True.
- If min_value is None, then max_value is treated as an upper bound
- If max_value is None, then min_value is treated as a lower bound

See also:

expect_column_mean_to_be_between

expect_column_stdev_to_be_between

expect_column_quantile_values_to_be_between(*column*, *quantile_ranges*, *allow_relative_error=False*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect specific provided column quantiles to be between provided minimum and maximum values.

`quantile_ranges` must be a dictionary with two keys:

- `quantiles`: (list of float) increasing ordered list of desired quantile values
- `value_ranges`: (list of lists): Each element in this list consists of a list with two values, a lower and upper bound (inclusive) for the corresponding quantile.

For each provided range:

- min_value and max_value are both inclusive.
- If min_value is None, then max_value is treated as an upper bound only
- If max_value is None, then min_value is treated as a lower bound only

The length of the quantiles list and quantile_values list must be equal.

For example:

```
# my_df.my_col = [1,2,2,3,3,3,4]
>>> my_df.expect_column_quantile_values_to_be_between(
    "my_col",
    {
        "quantiles": [0., 0.333, 0.6667, 1.],
        "value_ranges": [[0,1], [2,3], [3,4], [4,5]]
    }
)
{
  "success": True,
    "result": {
      "observed_value": {
        "quantiles: [0., 0.333, 0.6667, 1.],
        "values": [1, 2, 3, 4],
      }
      "element_count": 7,
      "missing_count": 0,
      "missing_percent": 0.0,
      "details": {
        "success_details": [true, true, true, true]
      }
    }
  }
}
```

expect_column_quantile_values_to_be_between can be computationally intensive for large datasets.

expect_column_quantile_values_to_be_between is a `column_aggregate_expectation`.

Parameters

- **column** (*str*) – The column name.
- **quantile_ranges** (*dictionary*) – Quantiles and associated value ranges for the column. See above for details.

- **allow_relative_error** (*boolean*) – Whether to allow relative error in quantile communications on backends that support or require it.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation: :: details.success_details

See also:

expect_column_min_to_be_between

expect_column_max_to_be_between

expect_column_median_to_be_between

expect_column_stdev_to_be_between(*column*, *min_value=None*, *max_value=None*, *strict_min=False*, *strict_max=False*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the column standard deviation to be between a minimum value and a maximum value. Uses sample standard deviation (normalized by N-1).

expect_column_stdev_to_be_between is a `column_aggregate_expectation`.

Parameters

- **column** (*str*) – The column name.
- **min_value** (*float or None*) – The minimum value for the column standard deviation.
- **max_value** (*float or None*) – The maximum value for the column standard deviation.
- **strict_min** (*boolean*) – If True, the column standard deviation must be strictly larger than min_value, default=False
- **strict_max** (*boolean*) – If True, the column standard deviation must be strictly smaller than max_value, default=False

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "observed_value": (float) The true standard deviation for the column
}
```

- min_value and max_value are both inclusive unless strict_min or strict_max are set to True.
- If min_value is None, then max_value is treated as an upper bound
- If max_value is None, then min_value is treated as a lower bound

See also:

expect_column_mean_to_be_between

expect_column_median_to_be_between

expect_column_unique_value_count_to_be_between(*column*, *min_value=None*, *max_value=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the number of unique values to be between a minimum value and a maximum value.

expect_column_unique_value_count_to_be_between is a `column_aggregate_expectation`.

Parameters

- **column** (*str*) – The column name.
- **min_value** (*int or None*) – The minimum number of unique values allowed.
- **max_value** (*int or None*) – The maximum number of unique values allowed.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.

- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "observed_value": (int) The number of unique values in the column
}
```

- min_value and max_value are both inclusive.
- If min_value is None, then max_value is treated as an upper bound
- If max_value is None, then min_value is treated as a lower bound

See also:

expect_column_proportion_of_unique_values_to_be_between

expect_column_proportion_of_unique_values_to_be_between(*column*, *min_value=0*, *max_value=1*, *strict_min=False*, *strict_max=False*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the proportion of unique values to be between a minimum value and a maximum value.

For example, in a column containing [1, 2, 2, 3, 3, 3, 4, 4, 4, 4], there are 4 unique values and 10 total values for a proportion of 0.4.

expect_column_proportion_of_unique_values_to_be_between is a `column_aggregate_expectation`.

Parameters

- **column** (*str*) – The column name.
- **min_value** (*float or None*) – The minimum proportion of unique values. (Proportions are on the range 0 to 1)
- **max_value** (*float or None*) – The maximum proportion of unique values. (Proportions are on the range 0 to 1)

- **strict_min** (*boolean*) – If True, the minimum proportion of unique values must be strictly larger than min_value, default=False
- **strict_max** (*boolean*) – If True, the maximum proportion of unique values must be strictly smaller than max_value, default=False

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "observed_value": (float) The proportion of unique values in the column
}
```

- min_value and max_value are both inclusive unless strict_min or strict_max are set to True.
- If min_value is None, then max_value is treated as an upper bound
- If max_value is None, then min_value is treated as a lower bound

See also:

expect_column_unique_value_count_to_be_between

expect_column_most_common_value_to_be_in_set(*column*, *value_set*, *ties_okay=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the most common value to be within the designated value set

expect_column_most_common_value_to_be_in_set is a `column_aggregate_expectation`.

Parameters

- **column** (*str*) – The column name
- **value_set** (*set-like*) – A list of potential values to match

Keyword Arguments **ties_okay** (*boolean or None*) – If True, then the expectation will still succeed if values outside the designated set are as common (but not more common) than designated values

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "observed_value": (list) The most common values in the column
}
```

observed_value contains a list of the most common values. Often, this will just be a single element. But if there's a tie for most common among multiple values, *observed_value* will contain a single copy of each most common value.

expect_column_sum_to_be_between (*column*, *min_value=None*, *max_value=None*, *strict_min=False*, *strict_max=False*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the column to sum to be between an min and max value

expect_column_sum_to_be_between is a `column_aggregate_expectation`.

Parameters

- **column** (*str*) – The column name
- **min_value** (*comparable type or None*) – The minimal sum allowed.
- **max_value** (*comparable type or None*) – The maximal sum allowed.
- **strict_min** (*boolean*) – If True, the minimal sum must be strictly larger than min_value, default=False
- **strict_max** (*boolean*) – If True, the maximal sum must be strictly smaller than max_value, default=False

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.

- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "observed_value": (list) The actual column sum
}
```

- min_value and max_value are both inclusive unless strict_min or strict_max are set to True.
- If min_value is None, then max_value is treated as an upper bound
- If max_value is None, then min_value is treated as a lower bound

expect_column_min_to_be_between(*column*, *min_value=None*, *max_value=None*, *strict_min=False*, *strict_max=False*, *parse_strings_as_datetimes=False*, *output_strftime_format=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the column minimum to be between an min and max value

expect_column_min_to_be_between is a `column_aggregate_expectation`.

Parameters

- **column** (*str*) – The column name
- **min_value** (*comparable type or None*) – The minimal column minimum allowed.
- **max_value** (*comparable type or None*) – The maximal column minimum allowed.
- **strict_min** (*boolean*) – If True, the minimal column minimum must be strictly larger than min_value, default=False
- **strict_max** (*boolean*) – If True, the maximal column minimum must be strictly smaller than max_value, default=False

Keyword Arguments

- **parse_strings_as_datetimes** (*Boolean or None*) – If True, parse min_value, max_values, and all non-null column values to datetimes before making comparisons.

- **output_strftime_format** (*str or None*) – A valid strfime format for datetime output. Only used if parse_strings_as_datetimes=True.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "observed_value": (list) The actual column min
}
```

- min_value and max_value are both inclusive unless strict_min or strict_max are set to True.
- If min_value is None, then max_value is treated as an upper bound
- If max_value is None, then min_value is treated as a lower bound

expect_column_max_to_be_between(*column*, *min_value=None*, *max_value=None*, *strict_min=False*, *strict_max=False*, *parse_strings_as_datetimes=False*, *output_strftime_format=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the column max to be between an min and max value

expect_column_max_to_be_between is a `column_aggregate_expectation`.

Parameters

- **column** (*str*) – The column name
- **min_value** (*comparable type or None*) – The minimum number of unique values allowed.
- **max_value** (*comparable type or None*) – The maximum number of unique values allowed.

Keyword Arguments

- **parse_strings_as_datetimes** (*Boolean or None*) – If True, parse min_value, max_values, and all non-null column values to datetimes before making comparisons.
- **output_strftime_format** (*str or None*) – A valid strfime format for datetime output. Only used if parse_strings_as_datetimes=True.
- **strict_min** (*boolean*) – If True, the minimal column minimum must be strictly larger than min_value, default=False
- **strict_max** (*boolean*) – If True, the maximal column minimum must be strictly smaller than max_value, default=False

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "observed_value": (list) The actual column max
}
```

- min_value and max_value are both inclusive unless strict_min or strict_max are set to True.
- If min_value is None, then max_value is treated as an upper bound
- If max_value is None, then min_value is treated as a lower bound

expect_column_chisquare_test_p_value_to_be_greater_than(*column*, *partition_object=None*, *p=0.05*, *tail_weight_holdout=0*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column values to be distributed similarly to the provided categorical partition. This expectation compares categorical distributions using a Chi-squared test. It returns *success=True* if values in the column match the distribution of the provided partition.

expect_column_chisquare_test_p_value_to_be_greater_than is a `column_aggregate_expectation`.

Parameters

- **column** (*str*) – The column name.
- **partition_object** (*dict*) – The expected partition object (see *Partition Objects*).
- **p** (*float*) – The p-value threshold for rejecting the null hypothesis of the Chi-Squared test. For values below the specified threshold, the expectation will return *success=False*, rejecting the null hypothesis that the distributions are the same. Defaults to 0.05.

Keyword Arguments **tail_weight_holdout** (*float between 0 and 1 or None*) – The amount of weight to split uniformly between values observed in the data but not present in the provided partition. tail_weight_holdout provides a mechanism to make the test less strict by assigning positive weights to unknown values observed in the data that are not present in the partition.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "observed_value": (float) The true p-value of the Chi-squared test
    "details": {
        "observed_partition" (dict):
            The partition observed in the data.
        "expected_partition" (dict):
            The partition expected from the data, after including tail_weight_
↪holdout
    }
}
```

expect_column_bootstrapped_ks_test_p_value_to_be_greater_than(*column*, *partition_object=None*, *p=0.05*, *bootstrap_samples=None*, *bootstrap_sample_size=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect column values to be distributed similarly to the provided continuous partition. This expectation compares continuous distributions using a bootstrapped Kolmogorov-Smirnov test. It returns *success=True* if values in the column match the distribution of the provided partition.

The expected cumulative density function (CDF) is constructed as a linear interpolation between the bins, using the provided weights. Consequently the test expects a piecewise uniform distribution using the bins from the provided partition object.

`expect_column_bootstrapped_ks_test_p_value_to_be_greater_than` is a `column_aggregate_expectation`.

Parameters

- **column** (*str*) – The column name.
- **partition_object** (*dict*) – The expected partition object (see *Partition Objects*).
- **p** (*float*) – The p-value threshold for the Kolmogorov-Smirnov test. For values below the specified threshold the expectation will return *success=False*, rejecting the null hypothesis that the distributions are the same. Defaults to 0.05.

Keyword Arguments

- **bootstrap_samples** (*int*) – The number bootstrap rounds. Defaults to 1000.
- **bootstrap_sample_size** (*int*) – The number of samples to take from the column for each bootstrap. A larger sample will increase the specificity of the test. Defaults to 2 * len(partition_object['weights'])

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "observed_value": (float) The true p-value of the KS test
    "details": {
        "bootstrap_samples": The number of bootstrap rounds used
        "bootstrap_sample_size": The number of samples taken from
            the column in each bootstrap round
        "observed_cdf": The cumulative density function observed
            in the data, a dict containing 'x' values and cdf_values
            (suitable for plotting)
        "expected_cdf" (dict):
            The cumulative density function expected based on the
            partition object, a dict containing 'x' values and
            cdf_values (suitable for plotting)
        "observed_partition" (dict):
            The partition observed on the data, using the provided
            bins but also expanding from min(column) to max(column)
        "expected_partition" (dict):
            The partition expected from the data. For KS test,
            this will always be the partition_object parameter
    }
}
```

expect_column_kl_divergence_to_be_less_than(*column*, *partition_object=None*, *threshold=None*, *tail_weight_holdout=0*, *internal_weight_holdout=0*, *bucketize_data=True*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the Kulback-Leibler (KL) divergence (relative entropy) of the specified column with respect to the partition object to be lower than the provided threshold.

KL divergence compares two distributions. The higher the divergence value (relative entropy), the larger the difference between the two distributions. A relative entropy of zero indicates that the data are distributed identically, *when binned according to the provided partition*.

In many practical contexts, choosing a value between 0.5 and 1 will provide a useful test.

This expectation works on both categorical and continuous partitions. See notes below for details.

`expect_column_kl_divergence_to_be_less_than` is a `column_aggregate_expectation`.

Parameters

- **column** (*str*) – The column name.
- **partition_object** (*dict*) – The expected partition object (see *Partition Objects*).
- **threshold** (*float*) – The maximum KL divergence to for which to return *success=True*. If KL divergence is larger than the provided threshold, the test will return *success=False*.

Keyword Arguments

- **internal_weight_holdout** (*float between 0 and 1 or None*) – The amount of weight to split uniformly among zero-weighted partition bins. internal_weight_holdout provides a mechanisms to make the test less strict by assigning positive weights to values observed in the data for which the partition explicitly expected zero

weight. With no internal_weight_holdout, any value observed in such a region will cause KL divergence to rise to +Infinity. Defaults to 0.

- **tail_weight_holdout** (*float between 0 and 1 or None*) – The amount of weight to add to the tails of the histogram. Tail weight holdout is split evenly between (-Infinity, min(partition_object['bins'])) and (max(partition_object['bins']), +Infinity). tail_weight_holdout provides a mechanism to make the test less strict by assigning positive weights to values observed in the data that are not present in the partition. With no tail_weight_holdout, any value observed outside the provided partition_object will cause KL divergence to rise to +Infinity. Defaults to 0.
- **bucketize_data** (*boolean*) – If True, then continuous data will be bucketized before evaluation. Setting this parameter to false allows evaluation of KL divergence with a None partition object for profiling against discrete data.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
  "observed_value": (float) The true KL divergence (relative entropy) or None␣
↪if the value is                    calculated as infinity, -infinity, or NaN
  "details": {
    "observed_partition": (dict) The partition observed in the data
    "expected_partition": (dict) The partition against which the data were␣
↪compared,
                              after applying specified weight holdouts.
  }
}
```

If the partition_object is categorical, this expectation will expect the values in column to also be categorical.

- If the column includes values that are not present in the partition, the tail_weight_holdout will be equally split among those values, providing a mechanism to weaken the strictness of the expectation (otherwise, relative entropy would immediately go to infinity).
- If the partition includes values that are not present in the column, the test will simply include zero weight for that value.

If the partition_object is continuous, this expectation will discretize the values in the column according to the bins specified in the partition_object, and apply the test to the resulting distribution.

- The internal_weight_holdout and tail_weight_holdout parameters provide a mechanism to weaken the expectation, since an expected weight of zero would drive relative entropy to be infinite if any data are observed in that interval.
- If internal_weight_holdout is specified, that value will be distributed equally among any intervals with weight zero in the partition_object.
- If tail_weight_holdout is specified, that value will be appended to the tails of the bins ((-Infinity, min(bins)) and (max(bins), Infinity).

If relative entropy/kl divergence goes to infinity for any of the reasons mentioned above, the observed value will be set to None. This is because inf, -inf, Nan, are not json serializable and cause some json parsers to crash when encountered. The python None token will be serialized to null in json.

See also:

expect_column_chisquare_test_p_value_to_be_greater_than

expect_column_bootstrapped_ks_test_p_value_to_be_greater_than

expect_column_pair_values_to_be_equal(*column_A*, *column_B*, *ignore_row_if='both_values_are_missing'*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the values in column A to be the same as column B.

Parameters

- **column_A** (*str*) – The first column name
- **column_B** (*str*) – The second column name

Keyword Arguments **ignore_row_if** (*str*) – "both_values_are_missing", "either_value_is_missing", "neither"

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_column_pair_values_A_to_be_greater_than_B(*column_A*, *column_B*, *or_equal=None*, *parse_strings_as_datetimes=False*, *allow_cross_type_comparisons=None*, *ignore_row_if='both_values_are_missing'*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect values in column A to be greater than column B.

Parameters

- **column_A** (*str*) – The first column name
- **column_B** (*str*) – The second column name
- **or_equal** (*boolean or None*) – If True, then values can be equal, not strictly greater

Keyword Arguments

- **allow_cross_type_comparisons** (*boolean or None*) – If True, allow comparisons between types (e.g. integer and string). Otherwise, attempting such comparisons will raise an exception.
- **ignore_row_if** (*str*) – "both_values_are_missing", "either_value_is_missing", "neither

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_column_pair_values_to_be_in_set(*column_A*, *column_B*, *value_pairs_set*, *ignore_row_if='both_values_are_missing'*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect paired values from columns A and B to belong to a set of valid pairs.

Parameters

- **column_A** (*str*) – The first column name
- **column_B** (*str*) – The second column name

- **value_pairs_set** (*list of tuples*) – All the valid pairs to be matched

Keyword Arguments **ignore_row_if** (*str*) – "both_values_are_missing", "either_value_is_missing", "never"

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_multicolumn_values_to_be_unique(*column_list*, *ignore_row_if='all_values_are_missing'*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the values for each row to be unique across the columns listed.

Parameters **column_list** (*tuple or list*) – The first column name

Keyword Arguments **ignore_row_if** (*str*) – "all_values_are_missing", "any_value_is_missing", "never"

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

11.2.2 PandasDataset

class great_expectations.dataset.pandas_dataset.**MetaPandasDataset**(**args*, ***kwargs*)

Bases: *great_expectations.dataset.dataset.Dataset*

MetaPandasDataset is a thin layer between Dataset and PandasDataset.

This two-layer inheritance is required to make @classmethod decorators work.

Practically speaking, that means that MetaPandasDataset implements expectation decorators, like *column_map_expectation* and *column_aggregate_expectation*, and PandasDataset implements the expectation methods themselves.

classmethod column_map_expectation(*func*)

Constructs an expectation using column-map semantics.

The MetaPandasDataset implementation replaces the "column" parameter supplied by the user with a pandas Series object containing the actual column from the relevant pandas dataframe. This simplifies the implementing expectation logic while preserving the standard Dataset signature and expected behavior.

See `column_map_expectation` for full documentation of this function.

classmethod column_pair_map_expectation(*func*)

The column_pair_map_expectation decorator handles boilerplate issues surrounding the common pattern of evaluating truthiness of some condition on a per row basis across a pair of columns.

classmethod multicolumn_map_expectation(*func*)

The multicolumn_map_expectation decorator handles boilerplate issues surrounding the common pattern of evaluating truthiness of some condition on a per row basis across a set of columns.

class great_expectations.dataset.pandas_dataset.**PandasDataset**(**args*, ***kwargs*)

Bases: *great_expectations.dataset.pandas_dataset.MetaPandasDataset*, `pandas.core.frame.DataFrame`

PandasDataset instantiates the great_expectations Expectations API as a subclass of a pandas.DataFrame.

For the full API reference, please see `Dataset`

Notes

1. Samples and Subsets of PandaDataSet have ALL the expectations of the original data frame unless the user specifies the `discard_subset_failing_expectations = True` property on the original data frame.
2. Concatenations, joins, and merges of PandaDataSets contain NO expectations (since no autoinspection is performed by default).

get_row_count()

Returns: int, table row count

get_column_count()

Returns: int, table column count

get_table_columns()

Returns: List[str], list of column names

get_column_sum(*column*)

Returns: float

get_column_max(*column*, *parse_strings_as_datetimes=False*)
Returns: any

get_column_min(*column*, *parse_strings_as_datetimes=False*)
Returns: any

get_column_mean(*column*)
Returns: float

get_column_nonnull_count(*column*)
Returns: int

get_column_value_counts(*column*, *sort='value'*, *collate=None*)
Get a series containing the frequency counts of unique values from the named column.

Parameters

- **column** – the column for which to obtain value_counts
- **sort** (*string*) – must be one of "value", "count", or "none". - if "value" then values in the resulting partition object will be sorted lexigraphically - if "count" then values will be sorted according to descending count (frequency) - if "none" then values will not be sorted
- **collate** (*string*) – the collate (sort) method to be used on supported backends (SqlAlchemy only)

Returns pd.Series of value counts for a column, sorted according to the value requested in sort

get_column_unique_count(*column*)
Returns: int

get_column_modes(*column*)
Returns: List[any], list of modes (ties OK)

get_column_median(*column*)
Returns: any

get_column_quantiles(*column*, *quantiles*, *allow_relative_error=False*)
Get the values in column closest to the requested quantiles :param column: name of column :type column: string :param quantiles: the quantiles to return. quantiles *must* be a tuple to ensure caching is possible :type quantiles: tuple of float

Returns the nearest values in the dataset to those quantiles

Return type List[any]

get_column_stdev(*column*)
Returns: float

get_column_hist(*column*, *bins*)
Get a histogram of column values :param column: the column for which to generate the histogram :param bins: the bins to slice the histogram. bins *must* be a tuple to ensure caching is possible :type bins: tuple

Returns: List[int], a list of counts corresponding to bins

get_column_count_in_range(*column*, *min_val=None*, *max_val=None*, *strict_min=False*, *strict_max=True*)
Returns: int

expect_column_values_to_not_match_regex_list(*column*, *regex_list*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the column entries to be strings that do not match any of a list of regular expressions. Matches can be anywhere in the string.

expect_column_values_to_not_match_regex_list is a *column_map_expectation*.

Args: column (str): The column name. regex_list (list): The list of regular expressions which the column entries should not match

Keyword Args: mostly (None or a float between 0 and 1): Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters: result_format (str or None): Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*. include_config (boolean): If True, then include the expectation config as part of the result object. For more detail, see *include_config*. catch_exceptions (boolean or None): If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*. meta (dict or None): A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns: A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See Also: *expect_column_values_to_match_regex_list*

expect_column_parameterized_distribution_ks_test_p_value_to_be_greater_than(*column*, *distribution*, *p_value=0.05*, *params=None*, *result_format=N*, *include_config=T*, *catch_exceptio*, *meta=None*)

Expect the column values to be distributed similarly to a scipy distribution. This expectation compares the provided column to the specified continuous distribution with a parametric Kolmogorov-Smirnov test. The K-S test compares the provided column to the cumulative density function (CDF) of the specified scipy distribution. If you don't know the desired distribution shape parameters, use the *ge.dataset.util.infer_distribution_parameters()* utility function to estimate them.

It returns 'success'=True if the p-value from the K-S test is greater than or equal to the provided p-value.

expect_column_parameterized_distribution_ks_test_p_value_to_be_greater_than is a *column_aggregate_expectation*.

Parameters

- **column** (*str*) – The column name.
- **distribution** (*str*) – The scipy distribution name. See: https://docs.scipy.org/doc/scipy/reference/stats.html Currently supported distributions are listed in the Notes section below.
- **p_value** (*float*) – The threshold p-value for a passing test. Default is 0.05.
- **params** (*dict or list*) – A dictionary or positional list of shape parameters that describe the distribution you want to test the data against. Include key values specific to

the distribution from the appropriate scipy distribution CDF function. 'loc' and 'scale' are used as translational parameters. See https://docs.scipy.org/doc/scipy/reference/stats.html#continuous-distributions

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

Notes

These fields in the result object are customized for this expectation:

```
{
    "details":
        "expected_params" (dict): The specified or inferred parameters of the␣
↪distribution to test                          against
        "ks_results" (dict): The raw result of stats.kstest()
}
```

- The Kolmogorov-Smirnov test's null hypothesis is that the column is similar to the provided distribution.
- Supported scipy distributions:
  - norm
  - beta
  - gamma
  - uniform
  - chi2
  - expon

expect_column_pair_values_to_be_equal(*column_A*, *column_B*, *ignore_row_if='both_values_are_missing'*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the values in column A to be the same as column B.

Parameters

- **column_A** (*str*) – The first column name

- **column_B** (*str*) – The second column name

Keyword Arguments **ignore_row_if** (*str*) – "both_values_are_missing", "either_value_is_missing", "neither"

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_column_pair_values_A_to_be_greater_than_B(*column_A*, *column_B*, *or_equal=None*, *parse_strings_as_datetimes=None*, *allow_cross_type_comparisons=None*, *ignore_row_if='both_values_are_missing'*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect values in column A to be greater than column B.

Parameters

- **column_A** (*str*) – The first column name
- **column_B** (*str*) – The second column name
- **or_equal** (*boolean or None*) – If True, then values can be equal, not strictly greater

Keyword Arguments

- **allow_cross_type_comparisons** (*boolean or None*) – If True, allow comparisons between types (e.g. integer and string). Otherwise, attempting such comparisons will raise an exception.
- **ignore_row_if** (*str*) – "both_values_are_missing", "either_value_is_missing", "neither

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.

- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_column_pair_values_to_be_in_set(*column_A*, *column_B*, *value_pairs_set*, *ignore_row_if='both_values_are_missing'*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect paired values from columns A and B to belong to a set of valid pairs.

Parameters

- **column_A** (*str*) – The first column name
- **column_B** (*str*) – The second column name
- **value_pairs_set** (*list of tuples*) – All the valid pairs to be matched

Keyword Arguments **ignore_row_if** (*str*) – "both_values_are_missing", "either_value_is_missing", "never"

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

expect_multicolumn_values_to_be_unique(*column_list*, *ignore_row_if='all_values_are_missing'*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)

Expect the values for each row to be unique across the columns listed.

Parameters **column_list** (*tuple or list*) – The first column name

Keyword Arguments **ignore_row_if** (*str*) – "all_values_are_missing", "any_value_is_missing", "never"

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.

- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

11.2.3 SqlAlchemyDataset

class great_expectations.dataset.sqlalchemy_dataset.**MetaSqlAlchemyDataset**(**args*, ***kwargs*)

Bases: *great_expectations.dataset.dataset.Dataset*

classmethod column_map_expectation(*func*)

For SqlAlchemy, this decorator allows individual column_map_expectations to simply return the filter that describes the expected condition on their data.

The decorator will then use that filter to obtain unexpected elements, relevant counts, and return the formatted object.

class great_expectations.dataset.sqlalchemy_dataset.**SqlAlchemyDataset**(*table_name=None*, *engine=None*, *connection_string=None*, *custom_sql=None*, *schema=None*, **args*, ***kwargs*)

Bases: *great_expectations.dataset.sqlalchemy_dataset.MetaSqlAlchemyDataset*

classmethod from_dataset(*dataset=None*)

This base implementation naively passes arguments on to the real constructor, which is suitable really when a constructor knows to take its own type. In general, this should be overridden

head(*n=5*)

Returns a *PandasDataset* with the first *n* rows of the given Dataset

get_row_count()

Returns: int, table row count

get_column_count()

Returns: int, table column count

get_table_columns()

Returns: List[str], list of column names

get_column_nonnull_count(*column*)

Returns: int

get_column_sum(*column*)
Returns: float

get_column_max(*column*, *parse_strings_as_datetimes=False*)
Returns: any

get_column_min(*column*, *parse_strings_as_datetimes=False*)
Returns: any

get_column_value_counts(*column*, *sort='value'*, *collate=None*)
Get a series containing the frequency counts of unique values from the named column.

Parameters

- **column** – the column for which to obtain value_counts
- **sort** (*string*) – must be one of "value", "count", or "none". - if "value" then values in the resulting partition object will be sorted lexigraphically - if "count" then values will be sorted according to descending count (frequency) - if "none" then values will not be sorted
- **collate** (*string*) – the collate (sort) method to be used on supported backends (SqlAlchemy only)

Returns pd.Series of value counts for a column, sorted according to the value requested in sort

get_column_mean(*column*)
Returns: float

get_column_unique_count(*column*)
Returns: int

get_column_median(*column*)
Returns: any

get_column_quantiles(*column*, *quantiles*, *allow_relative_error=False*)
Get the values in column closest to the requested quantiles :param column: name of column :type column: string :param quantiles: the quantiles to return. quantiles *must* be a tuple to ensure caching is possible :type quantiles: tuple of float

Returns the nearest values in the dataset to those quantiles

Return type List[any]

get_column_stdev(*column*)
Returns: float

get_column_hist(*column*, *bins*)
return a list of counts corresponding to bins

Parameters

- **column** – the name of the column for which to get the histogram
- **bins** – tuple of bin edges for which to get histogram values; *must* be tuple to support caching

get_column_count_in_range(*column*, *min_val=None*, *max_val=None*, *strict_min=False*, *strict_max=True*)
Returns: int

create_temporary_table(*table_name*, *custom_sql*, *schema_name=None*)
Create Temporary table based on sql query. This will be used as a basis for executing expectations. WARNING: this feature is new in v0.4. It hasn't been tested in all SQL dialects, and may change based on community feedback. :param custom_sql:

column_reflection_fallback()
If we can't reflect the table, use a query to at least get column names.

expect_column_values_to_not_match_regex_list(*column*, *regex_list*, *mostly=None*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)
Expect the column entries to be strings that do not match any of a list of regular expressions. Matches can be anywhere in the string.

expect_column_values_to_not_match_regex_list is a *column_map_expectation*.

Parameters

- **column** (*str*) – The column name.
- **regex_list** (*list*) – The list of regular expressions which the column entries should not match

Keyword Arguments **mostly** (*None or a float between 0 and 1*) – Return *"success": True* if at least mostly fraction of values match the expectation. For more detail, see *mostly*.

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.
- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

See also:

expect_column_values_to_match_regex_list

class great_expectations.dataset.sqlalchemy_dataset.**SqlAlchemyBatchReference**(*engine*, *table_name=None*, *schema=None*, *query=None*)

Bases: object

get_init_kwargs()

11.2.4 SparkDFDataset

class great_expectations.dataset.sparkdf_dataset.**MetaSparkDFDataset**(**args*, ***kwargs*)

Bases: *great_expectations.dataset.dataset.Dataset*

MetaSparkDFDataset is a thin layer between Dataset and SparkDFDataset. This two-layer inheritance is required to make @classmethod decorators work. Practically speaking, that means that MetaSparkDFDataset implements expectation decorators, like *column_map_expectation* and *column_aggregate_expectation*, and SparkDFDataset implements the expectation methods themselves.

classmethod column_map_expectation(*func*)

Constructs an expectation using column-map semantics.

The MetaSparkDFDataset implementation replaces the "column" parameter supplied by the user with a Spark Dataframe with the actual column data. The current approach for functions implementing expectation logic is to append a column named "__success" to this dataframe and return to this decorator.

See `column_map_expectation` for full documentation of this function.

classmethod column_pair_map_expectation(*func*)

The column_pair_map_expectation decorator handles boilerplate issues surrounding the common pattern of evaluating truthiness of some condition on a per row basis across a pair of columns.

class great_expectations.dataset.sparkdf_dataset.**SparkDFDataset**(*spark_df*, **args*, ***kwargs*)

Bases: *great_expectations.dataset.sparkdf_dataset.MetaSparkDFDataset*

This class holds an attribute *spark_df* which is a spark.sql.DataFrame.

classmethod from_dataset(*dataset=None*)

This base implementation naively passes arguments on to the real constructor, which is suitable really when a constructor knows to take its own type. In general, this should be overridden

head(*n=5*)

Returns a *PandasDataset* with the first *n* rows of the given Dataset

get_row_count()

Returns: int, table row count

get_column_count()

Returns: int, table column count

get_table_columns()

Returns: List[str], list of column names

get_column_nonnull_count(*column*)

Returns: int

get_column_mean(*column*)

Returns: float

get_column_sum(*column*)

Returns: float

get_column_max(*column*, *parse_strings_as_datetimes=False*)

Returns: any

get_column_min(*column*, *parse_strings_as_datetimes=False*)

Returns: any

get_column_value_counts(*column*, *sort='value'*, *collate=None*)
Get a series containing the frequency counts of unique values from the named column.

Parameters

- **column** – the column for which to obtain value_counts
- **sort** (*string*) – must be one of "value", "count", or "none". - if "value" then values in the resulting partition object will be sorted lexigraphically - if "count" then values will be sorted according to descending count (frequency) - if "none" then values will not be sorted
- **collate** (*string*) – the collate (sort) method to be used on supported backends (SqlAlchemy only)

Returns pd.Series of value counts for a column, sorted according to the value requested in sort

get_column_unique_count(*column*)
Returns: int

get_column_modes(*column*)
leverages computation done in _get_column_value_counts

get_column_median(*column*)
Returns: any

get_column_quantiles(*column*, *quantiles*, *allow_relative_error=False*)
Get the values in column closest to the requested quantiles :param column: name of column :type column: string :param quantiles: the quantiles to return. quantiles *must* be a tuple to ensure caching is possible :type quantiles: tuple of float

Returns the nearest values in the dataset to those quantiles

Return type List[any]

get_column_stdev(*column*)
Returns: float

get_column_hist(*column*, *bins*)
return a list of counts corresponding to bins

get_column_count_in_range(*column*, *min_val=None*, *max_val=None*, *strict_min=False*, *strict_max=True*)
Returns: int

expect_column_pair_values_to_be_equal(*column_A*, *column_B*, *ignore_row_if='both_values_are_missing'*, *result_format=None*, *include_config=True*, *catch_exceptions=None*, *meta=None*)
Expect the values in column A to be the same as column B.

Parameters

- **column_A** (*str*) – The first column name
- **column_B** (*str*) – The second column name

Keyword Arguments **ignore_row_if** (*str*) – "both_values_are_missing", "either_value_is_missing", "neither"

Other Parameters

- **result_format** (*str or None*) – Which output mode to use: *BOOLEAN_ONLY*, *BASIC*, *COMPLETE*, or *SUMMARY*. For more detail, see *result_format*.

- **include_config** (*boolean*) – If True, then include the expectation config as part of the result object. For more detail, see *include_config*.
- **catch_exceptions** (*boolean or None*) – If True, then catch exceptions and include them as part of the result object. For more detail, see *catch_exceptions*.
- **meta** (*dict or None*) – A JSON-serializable dictionary (nesting allowed) that will be included in the output without modification. For more detail, see *meta*.

Returns

A JSON-serializable expectation result object.

Exact fields vary depending on the values passed to *result_format* and *include_config*, *catch_exceptions*, and *meta*.

11.2.5 util

great_expectations.dataset.util.**is_valid_partition_object**(*partition_object*)
Tests whether a given object is a valid continuous or categorical partition object. :param partition_object: The partition_object to evaluate :return: Boolean

great_expectations.dataset.util.**is_valid_categorical_partition_object**(*partition_object*)
Tests whether a given object is a valid categorical partition object. :param partition_object: The partition_object to evaluate :return: Boolean

great_expectations.dataset.util.**is_valid_continuous_partition_object**(*partition_object*)
Tests whether a given object is a valid continuous partition object. See *Partition Objects*.

Parameters **partition_object** – The partition_object to evaluate

Returns Boolean

great_expectations.dataset.util.**categorical_partition_data**(*data*)
Convenience method for creating weights from categorical data.

Parameters **data** (*list-like*) – The data from which to construct the estimate.

Returns

A new partition object:

```
{
    "values": (list) The categorical values present in the data
    "weights": (list) The weights of the values in the partition.
}
```

See *Partition Objects*.

great_expectations.dataset.util.**kde_partition_data**(*data*, *estimate_tails=True*)
Convenience method for building a partition and weights using a gaussian Kernel Density Estimate and default bandwidth.

Parameters

- **data** (*list-like*) – The data from which to construct the estimate
- **estimate_tails** (*bool*) – Whether to estimate the tails of the distribution to keep the partition object finite

Returns

A new partition_object:

```
{
    "bins": (list) The endpoints of the partial partition of reals,
    "weights": (list) The densities of the bins implied by the␣
↪partition.
}

See :ref:`partition_object`.
```

great_expectations.dataset.util.**partition_data**(*data*, *bins='auto'*, *n_bins=10*)

great_expectations.dataset.util.**continuous_partition_data**(*data*, *bins='auto'*, *n_bins=10*, ***kwargs*)

Convenience method for building a partition object on continuous data

Parameters

- **data** (*list-like*) – The data from which to construct the estimate.
- **bins** (*string*) – One of 'uniform' (for uniformly spaced bins), 'ntile' (for percentile-spaced bins), or 'auto' (for automatically spaced bins)
- **n_bins** (*int*) – Ignored if bins is auto.
- **kwargs** (*mapping*) – Additional keyword arguments to be passed to numpy histogram

Returns

A new partition_object:

```
{
    "bins": (list) The endpoints of the partial partition of reals,
    "weights": (list) The densities of the bins implied by the␣
↪partition.
}
See :ref:`partition_object`.
```

great_expectations.dataset.util.**build_continuous_partition_object**(*dataset*, *column*, *bins='auto'*, *n_bins=10*, *allow_relative_error=False*)

Convenience method for building a partition object on continuous data from a dataset and column

Parameters

- **dataset** (*GE Dataset*) – the dataset for which to compute the partition
- **column** (*string*) – The name of the column for which to construct the estimate.
- **bins** (*string*) – One of 'uniform' (for uniformly spaced bins), 'ntile' (for percentile-spaced bins), or 'auto' (for automatically spaced bins)
- **n_bins** (*int*) – Ignored if bins is auto.
- **allow_relative_error** – passed to get_column_quantiles, set to False for only precise values, True to allow approximate values on systems with only binary choice (e.g. Redshift), and to a value between zero and one for systems that allow specification of relative error (e.g. SparkDFDataset).

Returns

A new partition_object:

```
{
    "bins": (list) The endpoints of the partial partition of reals,
    "weights": (list) The densities of the bins implied by the␣
↪partition.
}

See :ref:`partition_object`.
```

great_expectations.dataset.util.**build_categorical_partition_object**(*dataset*, *column*, *sort='value'*)

Convenience method for building a partition object on categorical data from a dataset and column

Parameters

- **dataset** (*GE Dataset*) – the dataset for which to compute the partition
- **column** (*string*) – The name of the column for which to construct the estimate.
- **sort** (*string*) – must be one of “value”, “count”, or “none”. - if “value” then values in the resulting partition object will be sorted lexigraphically - if “count” then values will be sorted according to descending count (frequency) - if “none” then values will not be sorted

Returns

A new partition_object:

```
{
    "values": (list) the categorical values for which each weight␣
↪applies,
    "weights": (list) The densities of the values implied by the␣
↪partition.
}
See :ref:`partition_object`.
```

great_expectations.dataset.util.**infer_distribution_parameters**(*data*, *distribution*, *params=None*)

Convenience method for determining the shape parameters of a given distribution

Parameters

- **data** (*list-like*) – The data to build shape parameters from.
- **distribution** (*string*) – Scipy distribution, determines which parameters to build.
- **params** (*dict or None*) – The known parameters. Parameters given here will not be altered. Keep as None to infer all necessary parameters from the data data.

Returns

A dictionary of named parameters:

```
{
    "mean": (float),
    "std_dev": (float),
    "loc": (float),
    "scale": (float),
    "alpha": (float),
    "beta": (float),
    "min": (float),
    "max": (float),
```

(continues on next page)

(continued from previous page)

```
    "df": (float)
}

See: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.
↪kstest.html#scipy.stats.kstest
```

great_expectations.dataset.util.**validate_distribution_parameters**(*distribution*, *params*)

Ensures that necessary parameters for a distribution are present and that all parameters are sensical.

If parameters necessary to construct a distribution are missing or invalid, this function raises ValueError with an informative description. Note that 'loc' and 'scale' are optional arguments, and that 'scale' must be positive.

Parameters

- **distribution** (*string*) – The scipy distribution name, e.g. normal distribution is 'norm'.
- **params** (*dict or list*) – The distribution shape parameters in a named dictionary or positional list form following the scipy cdf argument scheme.

  params={'mean': 40, 'std_dev': 5} or params=[40, 5]

Exceptions: ValueError: With an informative description, usually when necessary parameters are omitted or are invalid.

great_expectations.dataset.util.**create_multiple_expectations**(*df*, *columns*, *expectation_type*, **args*, ***kwargs*)

Creates an identical expectation for each of the given columns with the specified arguments, if any.

Parameters

- **df** (*great_expectations.dataset*) – A great expectations dataset object.
- **columns** (*list*) – A list of column names represented as strings.
- **expectation_type** (*string*) – The expectation type.

Raises

- **KeyError if the provided column does not exist.** –
- **AttributeError if the provided expectation type does not exist or df is not a valid great expectations dataset.** –

Returns A list of expectation results.

11.3 DataContext Module

11.3.1 DataContext

class great_expectations.data_context.**BaseDataContext**(*project_config*, *context_root_dir=None*)

Bases: object

This class implements most of the functionality of DataContext, with a few exceptions.

1. BaseDataContext does not attempt to keep its project_config in sync with a file on disc.
2. **BaseDataContext doesn't attempt to "guess" paths or objects types. Instead, that logic is pushed** into DataContext class.

Together, these changes make BaseDataContext class more testable.

PROFILING_ERROR_CODE_TOO_MANY_DATA_ASSETS = 2

PROFILING_ERROR_CODE_SPECIFIED_DATA_ASSETS_NOT_FOUND = 3

PROFILING_ERROR_CODE_NO_GENERATOR_FOUND = 4

PROFILING_ERROR_CODE_MULTIPLE_GENERATORS_FOUND = 5

UNCOMMITTED_DIRECTORIES = ['data_docs', 'validations']

GE_UNCOMMITTED_DIR = 'uncommitted'

BASE_DIRECTORIES = ['expectations', 'notebooks', 'plugins', 'uncommitted']

NOTEBOOK_SUBDIRECTORIES = ['pandas', 'spark', 'sql']

GE_DIR = 'great_expectations'

GE_YML = 'great_expectations.yml'

GE_EDIT_NOTEBOOK_DIR = 'uncommitted'

classmethod validate_config(*project_config*)

add_store(*store_name*, *store_config*)

Add a new Store to the DataContext and (for convenience) return the instantiated Store object.

Parameters

- **store_name** (*str*) – a key for the new Store in in self._stores
- **store_config** (*dict*) – a config for the Store to add

Returns store (Store)

add_validation_operator(*validation_operator_name*, *validation_operator_config*)

Add a new ValidationOperator to the DataContext and (for convenience) return the instantiated object.

Parameters

- **validation_operator_name** (*str*) – a key for the new ValidationOperator in in self._validation_operators
- **validation_operator_config** (*dict*) – a config for the ValidationOperator to add

Returns validation_operator (ValidationOperator)

get_docs_sites_urls(*resource_identifier=None*)

Get URLs for a resource for all data docs sites.

This function will return URLs for any configured site even if the sites have not been built yet.

Parameters **resource_identifier** – optional. It can be an identifier of ExpectationSuite's, ValidationResults and other resources that have typed identifiers. If not provided, the method will return the URLs of the index page.

Returns a list of URLs. Each item is the URL for the resource for a data docs site

open_data_docs(*resource_identifier=None*)

A stdlib cross-platform way to open a file in a browser.

Parameters **resource_identifier** – ExpectationSuiteIdentifier, ValidationResultIdentifier or any other type's identifier. The argument is optional - when not supplied, the method returns the URL of the index page.

property root_directory
The root directory for configuration objects in the data context; the location in which `great_expectations.yml` is located.

property plugins_directory
The directory in which custom plugin modules should be placed.

property stores
A single holder for all Stores in this context

property datasources
A single holder for all Datasources in this context

property expectations_store_name

get_config_with_variables_substituted(*config=None*)

save_config_variable(*config_variable_name*, *value*)
Save config variable value

Parameters
- **config_variable_name** – name of the property
- **value** – the value to save for the property

Returns None

get_available_data_asset_names(*datasource_names=None*, *generator_names=None*)
Inspect datasource and generators to provide available data_asset objects.

Parameters
- **datasource_names** – list of datasources for which to provide available data_asset_name objects. If None, return available data assets for all datasources.
- **generator_names** – list of generators for which to provide available data_asset_name objects.

Returns

Dictionary describing available data assets

```
{
  datasource_name: {
    generator_name: [ data_asset_1, data_asset_2, ... ]
    ...
  }
  ...
}
```

Return type data_asset_names (dict)

build_batch_kwargs(*datasource*, *generator*, *name=None*, *partition_id=None*, ***kwargs*)
Builds batch kwargs using the provided datasource, generator, and batch_parameters.

Parameters
- **datasource** (*str*) – the name of the datasource for which to build batch_kwargs
- **generator** (*str*) – the name of the generator to use to build batch_kwargs
- **name** (*str*) – an optional name batch_parameter
- ****kwargs** – additional batch_parameters

Returns BatchKwargs

get_batch(*batch_kwargs*, *expectation_suite_name*, *data_asset_type=None*, *batch_parameters=None*)
Build a batch of data using batch_kwargs, and return a DataAsset with expectation_suite_name attached. If batch_parameters are included, they will be available as attributes of the batch.

Parameters
- **batch_kwargs** – the batch_kwargs to use; must include a datasource key
- **expectation_suite_name** – The ExpectationSuite or the name of the expectation_suite to get
- **data_asset_type** – the type of data_asset to build, with associated expectation implementations. This can generally be inferred from the datasource.
- **batch_parameters** – optional parameters to store as the reference description of the batch. They should reflect parameters that would provide the passed BatchKwargs.

Returns DataAsset

run_validation_operator(*validation_operator_name*, *assets_to_validate*, *run_id=None*, *evaluation_parameters=None*, ***kwargs*)

Run a validation operator to validate data assets and to perform the business logic around validation that the operator implements.

Parameters
- **validation_operator_name** – name of the operator, as appears in the context's config file
- **assets_to_validate** – a list that specifies the data assets that the operator will validate. The members of the list can be either batches, or a tuple that will allow the operator to fetch the batch: (batch_kwargs, expectation_suite_name)
- **run_id** – The run_id for the validation; if None, a default value will be used
- ****kwargs** – Additional kwargs to pass to the validation operator

Returns ValidationOperatorResult

list_validation_operator_names()

add_datasource(*name*, *initialize=True*, ***kwargs*)

Add a new datasource to the data context, with configuration provided as kwargs. :param name: the name for the new datasource to add :param initialize: if False, add the datasource to the config, but do not

initialize it, for example if a user needs to debug database connectivity.

Parameters **kwargs** (*keyword arguments*) – the configuration for the new datasource

Returns datasource (Datasource)

add_generator(*datasource_name*, *generator_name*, *class_name*, ***kwargs*)

Add a generator to the named datasource, using the provided configuration.

Parameters
- **datasource_name** – name of datasource to which to add the new generator
- **generator_name** – name of the generator to add
- **class_name** – class of the generator to add
- ****kwargs** – generator configuration, provided as kwargs

Returns:

get_config()

get_datasource(*datasource_name='default'*)

Get the named datasource

Parameters **datasource_name** (*str*) – the name of the datasource from the configuration

Returns datasource (Datasource)

list_expectation_suites()

Return a list of available expectation suite names.

list_datasources()

List currently-configured datasources on this context.

Returns each dictionary includes "name" and "class_name" keys
Return type List(dict)

create_expectation_suite(*expectation_suite_name*, *overwrite_existing=False*)
Build a new expectation suite and save it into the data_context expectation store.
Parameters
- **expectation_suite_name** – The name of the expectation_suite to create
- **overwrite_existing** (*boolean*) – Whether to overwrite expectation suite if expectation suite with given name already exists.

Returns A new (empty) expectation suite.

get_expectation_suite(*expectation_suite_name*)
Get a named expectation suite for the provided data_asset_name.
Parameters **expectation_suite_name** (*str*) – the name for the expectation suite
Returns expectation_suite

save_expectation_suite(*expectation_suite*, *expectation_suite_name=None*)
Save the provided expectation suite into the DataContext.
Parameters
- **expectation_suite** – the suite to save
- **expectation_suite_name** – the name of this expectation suite. If no name is provided the name will be read from the suite

Returns None

store_validation_result_metrics(*requested_metrics*, *validation_results*, *target_store_name*)

store_evaluation_parameters(*validation_results*, *target_store_name=None*)

property evaluation_parameter_store

property evaluation_parameter_store_name

property validations_store_name

property validations_store

get_validation_result(*expectation_suite_name*, *run_id=None*, *batch_identifier=None*, *validations_store_name=None*, *failed_only=False*)
Get validation results from a configured store.
Parameters
- **data_asset_name** – name of data asset for which to get validation result
- **expectation_suite_name** – expectation_suite name for which to get validation result (default: "default")
- **run_id** – run_id for which to get validation result (if None, fetch the latest result by alphanumeric sort)
- **validations_store_name** – the name of the store from which to get validation results
- **failed_only** – if True, filter the result to return only failed expectations

Returns validation_result

update_return_obj(*data_asset*, *return_obj*)
Helper called by data_asset.
Parameters
- **data_asset** – The data_asset whose validation produced the current return object
- **return_obj** – the return object to update

Returns the return object, potentially changed into a widget by the configured expectation explorer

Return type return_obj

build_data_docs(*site_names=None*, *resource_identifiers=None*)

Build Data Docs for your project.

These make it simple to visualize data quality in your project. These include Expectations, Validations & Profiles. The are built for all Datasources from JSON artifacts in the local repo including validations & profiles from the uncommitted directory.

Parameters

- **site_names** – if specified, build data docs only for these sites, otherwise, build all the sites specified in the context's config
- **resource_identifiers** – a list of resource identifiers (ExpectationSuiteIdentifier, ValidationResultIdentifier). If specified, rebuild HTML (or other views the data docs sites are rendering) only for the resources in this list. This supports incremental build of data docs sites (e.g., when a new validation result is created) and avoids full rebuild.

Returns A dictionary with the names of the updated data documentation sites as keys and the the location info of their index.html files as values

profile_datasource(*datasource_name*, *generator_name=None*, *data_assets=None*, *max_data_assets=20*, *profile_all_data_assets=True*, *profiler=<class 'great_expectations.profile.basic_dataset_profiler.BasicDatasetProfiler'>*, *dry_run=False*, *run_id='profiling'*, *additional_batch_kwargs=None*)

Profile the named datasource using the named profiler.

Parameters

- **datasource_name** – the name of the datasource for which to profile data_assets
- **generator_name** – the name of the generator to use to get batches
- **data_assets** – list of data asset names to profile
- **max_data_assets** – if the number of data assets the generator yields is greater than this max_data_assets, profile_all_data_assets=True is required to profile all
- **profile_all_data_assets** – when True, all data assets are profiled, regardless of their number
- **profiler** – the profiler class to use
- **dry_run** – when true, the method checks arguments and reports if can profile or specifies the arguments that are missing
- **additional_batch_kwargs** – Additional keyword arguments to be provided to get_batch when loading the data asset.

Returns

A dictionary:

```
{
    "success": True/False,
    "results": List of (expectation_suite, EVR) tuples for␣
↪each of the data_assets found in the datasource
}
```

When success = False, the error details are under "error" key

profile_data_asset(*datasource_name*, *generator_name=None*, *data_asset_name=None*, *batch_kwargs=None*, *expectation_suite_name=None*, *profiler=<class 'great_expectations.profile.basic_dataset_profiler.BasicDatasetProfiler'>*, *run_id='profiling'*, *additional_batch_kwargs=None*)

Profile a data asset

Parameters

- **datasource_name** – the name of the datasource to which the profiled data asset belongs
- **generator_name** – the name of the generator to use to get batches (only if batch_kwargs are not provided)
- **data_asset_name** – the name of the profiled data asset
- **batch_kwargs** – optional - if set, the method will use the value to fetch the batch to be profiled. If not passed, the generator (generator_name arg) will choose a batch
- **profiler** – the profiler class to use
- **run_id** – optional - if set, the validation result created by the profiler will be under the provided run_id
- **additional_batch_kwargs** –

:returns A dictionary:

```
{
    "success": True/False,
    "results": List of (expectation_suite, EVR) tuples for each of␣
↪the data_assets found in the datasource
}
```

When success = False, the error details are under "error" key

class great_expectations.data_context.**DataContext**(*context_root_dir=None*)

Bases: great_expectations.data_context.data_context.BaseDataContext

A DataContext represents a Great Expectations project. It organizes storage and access for expectation suites, datasources, notification settings, and data fixtures.

The DataContext is configured via a yml file stored in a directory called great_expectations; the configuration file as well as managed expectation suites should be stored in version control.

Use the *create* classmethod to create a new empty config, or instantiate the DataContext by passing the path to an existing data context root directory.

DataContexts use data sources you're already familiar with. Generators help introspect data stores and data execution frameworks (such as airflow, Nifi, dbt, or dagster) to describe and produce batches of data ready for analysis. This enables fetching, validation, profiling, and documentation of your data in a way that is meaningful within your existing infrastructure and work environment.

DataContexts use a datasource-based namespace, where each accessible type of data has a three-part normalized *data_asset_name*, consisting of *datasource/generator/generator_asset*.

- The datasource actually connects to a source of materialized data and returns Great Expectations DataAssets connected to a compute environment and ready for validation.
- The Generator knows how to introspect datasources and produce identifying "batch_kwargs" that define particular slices of data.
- The generator_asset is a specific name – often a table name or other name familiar to users – that generators can slice into batches.

An expectation suite is a collection of expectations ready to be applied to a batch of data. Since in many projects it is useful to have different expectations evaluate in different contexts–profiling vs. testing; warning vs. error; high vs. low compute; ML model or dashboard–suites provide a namespace option for selecting which expectations a DataContext returns.

In many simple projects, the datasource or generator name may be omitted and the DataContext will infer the correct name when there is no ambiguity.

Similarly, if no expectation suite name is provided, the DataContext will assume the name "default".

classmethod create(*project_root_dir=None*)
Build a new great_expectations directory and DataContext object in the provided project_root_dir.

create will not create a new "great_expectations" directory in the provided folder, provided one does not already exist. Then, it will initialize a new DataContext in that folder and write the resulting config.

Parameters **project_root_dir** – path to the root directory in which to create a new great_expectations directory
Returns DataContext

classmethod all_uncommitted_directories_exist(*ge_dir*)
Check if all uncommitted direcotries exist.

classmethod config_variables_yml_exist(*ge_dir*)
Check if all config_variables.yml exists.

classmethod write_config_variables_template_to_disk(*uncommitted_dir*)

classmethod write_project_template_to_disk(*ge_dir*)

classmethod scaffold_directories(*base_dir*)
Safely create GE directories for a new project.

classmethod scaffold_custom_data_docs(*plugins_dir*)
Copy custom data docs templates

classmethod scaffold_notebooks(*base_dir*)
Copy template notebooks into the notebooks directory for a project.

list_expectation_suite_names()
Lists the available expectation suite names

add_store(*store_name*, *store_config*)
Add a new Store to the DataContext and (for convenience) return the instantiated Store object.

Parameters
- **store_name** (*str*) – a key for the new Store in in self._stores
- **store_config** (*dict*) – a config for the Store to add

Returns store (Store)

add_datasource(*name*, ***kwargs*)
Add a new datasource to the data context, with configuration provided as kwargs. :param name: the name for the new datasource to add :param initialize: if False, add the datasource to the config, but do not
initialize it, for example if a user needs to debug database connectivity.

Parameters **kwargs** (*keyword arguments*) – the configuration for the new datasource
Returns datasource (Datasource)

classmethod find_context_root_dir()

classmethod find_context_yml_file(*search_start_dir=None*)
Search for the yml file starting here and moving upward.

classmethod does_config_exist_on_disk(*context_root_dir*)
Return True if the great_expectations.yml exists on disk.

classmethod is_project_initialized(*ge_dir*)
Return True if the project is initialized.

To be considered initialized, all of the following must be true: - all project directories exist (including uncommitted directories) - a valid great_expectations.yml is on disk - a config_variables.yml is on disk - the project has at least one datasource - the project has at least one suite

classmethod does_project_have_a_datasource_in_config_file(*ge_dir*)

great_expectations.data_context.util.**safe_mmkdir**(*directory*, *exist_ok=True*)
Simple wrapper since exist_ok is not available in python 2

great_expectations.data_context.util.**load_class**(*class_name*, *module_name*)
Dynamically load a class from strings or raise a helpful error.

great_expectations.data_context.util.**instantiate_class_from_config**(*config*, *runtime_environment*, *config_defaults=None*)
Build a GE class from configuration dictionaries.

great_expectations.data_context.util.**format_dict_for_error_message**(*dict_*)

great_expectations.data_context.util.**substitute_config_variable**(*template_str*, *config_variables_dict*)
This method takes a string, and if it contains a pattern ${SOME_VARIABLE} or $SOME_VARIABLE, returns a string where the pattern is replaced with the value of SOME_VARIABLE, otherwise returns the string unchanged.

If the environment variable SOME_VARIABLE is set, the method uses its value for substitution. If it is not set, the value of SOME_VARIABLE is looked up in the config variables store (file). If it is not found there, the input string is returned as is.

Parameters

- **template_str** – a string that might or might not be of the form ${SOME_VARIABLE} or $SOME_VARIABLE
- **config_variables_dict** – a dictionary of config variables. It is loaded from the config variables store (by default, "uncommitted/config_variables.yml file)

Returns

great_expectations.data_context.util.**substitute_all_config_variables**(*data*, *replace_variables_dict*)
Substitute all config variables of the form ${SOME_VARIABLE} in a dictionary-like config object for their values.

The method traverses the dictionary recursively.

Parameters

- **data** –

- **replace_variables_dict** –

Returns a dictionary with all the variables replaced with their values

great_expectations.data_context.util.**file_relative_path**(*dunderfile*, *relative_path*)

This function is useful when one needs to load a file that is relative to the position of the current file. (Such as when you encode a configuration file path in source file and want in runnable in any current working directory)

It is meant to be used like the following: file_relative_path(__file__, 'path/relative/to/file')

H/T https://github.com/dagster-io/dagster/blob/8a250e9619a49e8bff8e9aa7435df89c2d2ea039/python_modules/dagster/dagster/utils/__init__.py#L34

11.4 Datasource Module

class great_expectations.datasource.**Datasource**(*name*, *data_context=None*, *data_asset_type=None*, *generators=None*, ***kwargs*)

A Datasource connects to a compute environment and one or more storage environments and produces batches of data that Great Expectations can validate in that compute environment.

Each Datasource provides Batches connected to a specific compute environment, such as a SQL database, a Spark cluster, or a local in-memory Pandas DataFrame.

Datasources use Batch Kwargs to specify instructions for how to access data from relevant sources such as an existing object from a DAG runner, a SQL database, S3 bucket, or local filesystem.

To bridge the gap between those worlds, Datasources interact closely with *generators* which are aware of a source of data and can produce produce identifying information, called "batch_kwargs" that datasources can use to get individual batches of data. They add flexibility in how to obtain data such as with time-based partitioning, downsampling, or other techniques appropriate for the datasource.

For example, a generator could produce a SQL query that logically represents "rows in the Events table with a timestamp on February 7, 2012," which a SqlAlchemyDatasource could use to materialize a SqlAlchemyDataset corresponding to that batch of data and ready for validation.

Since opinionated DAG managers such as airflow, dbt, prefect.io, dagster can also act as datasources and/or generators for a more generic datasource.

When adding custom expectations by subclassing an existing DataAsset type, use the data_asset_type parameter to configure the datasource to load and return DataAssets of the custom type.

recognized_batch_parameters = {'limit'}

classmethod from_configuration(***kwargs*)

Build a new datasource from a configuration dictionary.

Parameters ****kwargs** – configuration key-value pairs

Returns the newly-created datasource

Return type datasource (*Datasource*)

classmethod build_configuration(*class_name*, *module_name='great_expectations.datasource'*, *data_asset_type=None*, *generators=None*, ***kwargs*)

Build a full configuration object for a datasource, potentially including generators with defaults.

Parameters

- **class_name** – The name of the class for which to build the config
- **module_name** – The name of the module in which the datasource class is located

- **data_asset_type** – A ClassConfig dictionary
- **generators** – Generator configuration dictionary
- ****kwargs** – Additional kwargs to be part of the datasource constructor's initialization

Returns A complete datasource configuration.

property name
Property for datasource name

property config

property data_context
Property for attached DataContext

add_generator(*name*, *class_name*, ***kwargs*)
Add a generator to the datasource.

Parameters

- **name** (*str*) – the name of the new generator to add
- **class_name** – class of the generator to add
- **kwargs** – additional keyword arguments will be passed directly to the new generator's constructor

Returns generator (Generator)

get_generator(*generator_name*)
Get the (named) generator from a datasource)

Parameters **generator_name** (*str*) – name of generator (default value is 'default')

Returns generator (Generator)

list_generators()
List currently-configured generators for this datasource.

Returns each dictionary includes "name" and "type" keys

Return type List(dict)

process_batch_parameters(*limit=None*, *dataset_options=None*)
Use datasource-specific configuration to translate any batch parameters into batch kwargs at the datasource level.

Parameters

- **limit** (*int*) – a parameter all datasources must accept to allow limiting a batch to a smaller number of rows.
- **dataset_options** (*dict*) – a set of kwargs that will be passed to the constructor of a dataset built using these batch_kwargs

Returns Result will include both parameters passed via argument and configured parameters.

Return type batch_kwargs

get_batch(*batch_kwargs*, *batch_parameters=None*)
Get a batch of data from the datasource.

Parameters

- **batch_kwargs** – the BatchKwargs to use to construct the batch

- **batch_parameters** – optional parameters to store as the reference description of the batch. They should reflect parameters that would provide the passed BatchKwargs.

Returns Batch

get_available_data_asset_names(*generator_names=None*)

Returns a dictionary of data_asset_names that the specified generator can provide. Note that some generators may not be capable of describing specific named data assets, and some generators (such as filesystem glob generators) require the user to configure data asset names.

Parameters **generator_names** – the generators for which to get available data asset names.

Returns

```
{
  generator_name: {
    names: [ (data_asset_1, data_asset_1_type), (data_asset_2, data_
↪asset_2_type) ... ]
  }
  ...
}
```

Return type dictionary consisting of sets of generator assets available for the specified generators

build_batch_kwargs(*generator*, *name=None*, *partition_id=None*, ***kwargs*)

11.4.1 PandasDatasource

class great_expectations.datasource.pandas_datasource.**PandasDatasource**(*name='pandas'*, *data_context=None*, *data_asset_type=None*, *generators=None*, *boto3_options=None*, *reader_method=None*, *reader_options=None*, *limit=None*, ***kwargs*)

Bases: great_expectations.datasource.datasource.Datasource

The PandasDatasource produces PandasDataset objects and supports generators capable of interacting with the local filesystem (the default subdir_reader generator), and from existing in-memory dataframes.

recognized_batch_parameters = {'dataset_options', 'limit', 'reader_method', 'reader_op

classmethod build_configuration(*data_asset_type=None*, *generators=None*, *boto3_options=None*, *reader_method=None*, *reader_options=None*, *limit=None*, ***kwargs*)

Build a full configuration object for a datasource, potentially including generators with defaults.

Parameters

- **data_asset_type** – A ClassConfig dictionary
- **generators** – Generator configuration dictionary
- **boto3_options** – Optional dictionary with key-value pairs to pass to boto3 during instantiation.

- **reader_method** – Optional default reader_method for generated batches
- **reader_options** – Optional default reader_options for generated batches
- **limit** – Optional default limit for generated batches
- ****kwargs** – Additional kwargs to be part of the datasource constructor's initialization

Returns A complete datasource configuration.

process_batch_parameters(*reader_method=None*, *reader_options=None*, *limit=None*, *dataset_options=None*)

Use datasource-specific configuration to translate any batch parameters into batch kwargs at the datasource level.

Parameters

- **limit** (*int*) – a parameter all datasources must accept to allow limiting a batch to a smaller number of rows.
- **dataset_options** (*dict*) – a set of kwargs that will be passed to the constructor of a dataset built using these batch_kwargs

Returns Result will include both parameters passed via argument and configured parameters.

Return type batch_kwargs

get_batch(*batch_kwargs*, *batch_parameters=None*)

Get a batch of data from the datasource.

Parameters

- **batch_kwargs** – the BatchKwargs to use to construct the batch
- **batch_parameters** – optional parameters to store as the reference description of the batch. They should reflect parameters that would provide the passed BatchKwargs.

Returns Batch

static guess_reader_method_from_path(*path*)

11.4.2 SqlAlchemyDatasource

class great_expectations.datasource.sqlalchemy_datasource.**SqlAlchemyDatasource**(*name='default'*, *data_context=No*, *data_asset_type=*, *cre-*, *den-*, *tials=None*, *gen-*, *er-*, *a-*, *tors=None*, ***kwargs*)

Bases: great_expectations.datasource.datasource.Datasource

A SqlAlchemyDatasource will provide data_assets converting batch_kwargs using the following rules:

- if the batch_kwargs include a table key, the datasource will provide a dataset object connected to that table

- if the batch_kwargs include a query key, the datasource will create a temporary table using that that query. The query can be parameterized according to the standard python Template engine, which uses $parameter, with additional kwargs passed to the get_batch method.

recognized_batch_parameters = {'dataset_options', 'limit', 'query_parameters'}

classmethod build_configuration(*data_asset_type=None*, *generators=None*, ***kwargs*)
Build a full configuration object for a datasource, potentially including generators with defaults.

Parameters

- **data_asset_type** – A ClassConfig dictionary
- **generators** – Generator configuration dictionary
- ****kwargs** – Additional kwargs to be part of the datasource constructor's initialization

Returns A complete datasource configuration.

get_batch(*batch_kwargs*, *batch_parameters=None*)
Get a batch of data from the datasource.

Parameters

- **batch_kwargs** – the BatchKwargs to use to construct the batch
- **batch_parameters** – optional parameters to store as the reference description of the batch. They should reflect parameters that would provide the passed BatchKwargs.

Returns Batch

process_batch_parameters(*query_parameters=None*, *limit=None*, *dataset_options=None*)
Use datasource-specific configuration to translate any batch parameters into batch kwargs at the datasource level.

Parameters

- **limit** (*int*) – a parameter all datasources must accept to allow limiting a batch to a smaller number of rows.
- **dataset_options** (*dict*) – a set of kwargs that will be passed to the constructor of a dataset built using these batch_kwargs

Returns Result will include both parameters passed via argument and configured parameters.

Return type batch_kwargs

11.4.3 SparkDFDatasource

class great_expectations.datasource.sparkdf_datasource.**SparkDFDatasource**(*name='default'*, *data_context=None*, *data_asset_type=None*, *generators=None*, *spark_config=None*, ***kwargs*)

Bases: great_expectations.datasource.datasource.Datasource

The SparkDFDatasource produces SparkDFDatasets and supports generators capable of interacting with local filesystem (the default subdir_reader generator) and databricks notebooks.

Accepted Batch Kwargs:

- PathBatchKwargs ("path" or "s3" keys)
- InMemoryBatchKwargs ("dataset" key)
- QueryBatchKwargs ("query" key)

recognized_batch_parameters = {'dataset_options', 'limit', 'reader_method', 'reader_op

classmethod build_configuration(*data_asset_type=None*, *generators=None*, *spark_config=None*, ***kwargs*)

Build a full configuration object for a datasource, potentially including generators with defaults.

Parameters

- **data_asset_type** – A ClassConfig dictionary
- **generators** – Generator configuration dictionary
- **spark_config** – dictionary of key-value pairs to pass to the spark builder
- ****kwargs** – Additional kwargs to be part of the datasource constructor's initialization

Returns A complete datasource configuration.

process_batch_parameters(*reader_method=None*, *reader_options=None*, *limit=None*, *dataset_options=None*)

Use datasource-specific configuration to translate any batch parameters into batch kwargs at the datasource level.

Parameters

- **limit** (*int*) – a parameter all datasources must accept to allow limiting a batch to a smaller number of rows.
- **dataset_options** (*dict*) – a set of kwargs that will be passed to the constructor of a dataset built using these batch_kwargs

Returns Result will include both parameters passed via argument and configured parameters.

Return type batch_kwargs

get_batch(*batch_kwargs*, *batch_parameters=None*)

class-private implementation of get_data_asset

static guess_reader_method_from_path(*path*)

11.5 Generator Module

class great_expectations.datasource.generator.batch_kwargs_generator.**BatchKwargsGenerator**(*n*
d
s

BatchKwargsGenerators produce identifying information, called "batch_kwargs" that datasources can use to get individual batches of data. They add flexibility in how to obtain data such as with time-based partitioning, downsampling, or other techniques appropriate for the datasource.

For example, a generator could produce a SQL query that logically represents "rows in the Events table with a timestamp on February 7, 2012," which a SqlAlchemyDatasource could use to materialize a SqlAlchemyDataset corresponding to that batch of data and ready for validation.

A batch is a sample from a data asset, sliced according to a particular rule. For example, an hourly slide of the Events table or "most recent *users* records."

A Batch is the primary unit of validation in the Great Expectations DataContext. Batches include metadata that identifies how they were constructed–the same "batch_kwargs" assembled by the generator, While not every datasource will enable re-fetching a specific batch of data, GE can store snapshots of batches or store metadata from an external data version control system.

Example Generator Configurations follow:

```
my_datasource_1:
  class_name: PandasDatasource
  generators:
    # This generator will provide two data assets, corresponding to the globs␣
↪defined under the "file_logs"
    # and "data_asset_2" keys. The file_logs asset will be partitioned according␣
↪to the match group
    # defined in partition_regex
    default:
      class_name: GlobReaderBatchKwargsGenerator
      base_directory: /var/logs
      reader_options:
        sep: "
      globs:
        file_logs:
          glob: logs/*.gz
          partition_regex: logs/file_(\d{0,4})_\.log\.gz
        data_asset_2:
          glob: data/*.csv

my_datasource_2:
  class_name: PandasDatasource
  generators:
    # This generator will create one data asset per subdirectory in /data
    # Each asset will have partitions corresponding to the filenames in that␣
↪subdirectory
    default:
      class_name: SubdirReaderBatchKwargsGenerator
      reader_options:
        sep: "
      base_directory: /data

my_datasource_3:
  class_name: SqlalchemyDatasource
  generators:
    # This generator will search for a file named with the name of the requested␣
↪generator asset and the
    # .sql suffix to open with a query to use to generate data
     default:
        class_name: QueryBatchKwargsGenerator
```

recognized_batch_parameters = {}

property name

get_available_data_asset_names()
Return the list of asset names known by this generator.

Returns A list of available names

get_available_partition_ids(*generator_asset*)
Applies the current _partitioner to the batches available on generator_asset and returns a list of valid partition_id strings that can be used to identify batches of data.

Parameters **generator_asset** – the generator asset whose partitions should be returned.

Returns A list of partition_id strings

get_config()

reset_iterator(*generator_asset*, ***kwargs*)

get_iterator(*generator_asset*, ***kwargs*)

build_batch_kwargs(*name=None*, *partition_id=None*, ***kwargs*)
The key workhorse. Docs forthcoming.

yield_batch_kwargs(*generator_asset*, ***kwargs*)

11.5.1 InMemoryGenerator

11.5.2 QueryBatchKwargsGenerator

class great_expectations.datasource.generator.query_generator.**QueryBatchKwargsGenerator**(*nam dat sou que que*

Bases: *great_expectations.datasource.generator.batch_kwargs_generator.BatchKwargsGenerator*

Produce query-style batch_kwargs from sql files stored on disk

recognized_batch_parameters = {'name', 'partition_id', 'query_parameters'}

add_query(*generator_asset*, *query*)

get_available_data_asset_names()
Return the list of asset names known by this generator.

Returns A list of available names

get_available_partition_ids(*generator_asset*)
Applies the current _partitioner to the batches available on generator_asset and returns a list of valid partition_id strings that can be used to identify batches of data.

Parameters **generator_asset** – the generator asset whose partitions should be returned.

Returns A list of partition_id strings

11.5.3 TableBatchKwargsGenerator

class great_expectations.datasource.generator.table_generator.**TableBatchKwargsGenerator**(*nam dat sou as- sets*

Bases: *great_expectations.datasource.generator.batch_kwargs_generator.BatchKwargsGenerator*

Provide access to already materialized tables or views in a database.

TableBatchKwargsGenerator can be used to define specific data asset names that take and substitute parameters, for example to support referring to the same data asset but with different schemas depending on provided batch_kwargs.

The python template language is used to substitute table name portions. For example, consider the following configurations:

```
my_generator:
  class_name: TableBatchKwargsGenerator
  assets:
    my_table:
      schema: $schema
      table: my_table
```

In that case, the asset my_datasource/my_generator/my_asset will refer to a table called my_table in a schema defined in batch_kwargs.

recognized_batch_parameters = {'limit', 'name', 'offset', 'query_parameters'}

get_available_data_asset_names()
Return the list of asset names known by this generator.

Returns A list of available names

get_available_partition_ids(*generator_asset*)
Applies the current _partitioner to the batches available on generator_asset and returns a list of valid partition_id strings that can be used to identify batches of data.

Parameters **generator_asset** – the generator asset whose partitions should be returned.

Returns A list of partition_id strings

11.5.4 SubdirReaderBatchKwargsGenerator

class great_expectations.datasource.generator.subdir_reader_generator.**SubdirReaderBatchKwa**

Bases: *great_expectations.datasource.generator.batch_kwargs_generator.BatchKwargsGenerator*

The SubdirReaderBatchKwargsGenerator inspects a filesystem and produces path-based batch_kwargs.

SubdirReaderBatchKwargsGenerator recognizes generator_assets using two criteria:

- for files directly in 'base_directory' with recognized extensions (.csv, .tsv, .parquet, .xls, .xlsx, .json), it uses the name of the file without the extension
- for other files or directories in 'base_directory', is uses the file or directory name

SubdirReaderBatchKwargsGenerator sees all files inside a directory of base_directory as batches of one datasource.

SubdirReaderBatchKwargsGenerator can also include configured reader_options which will be added to batch_kwargs generated by this generator.

recognized_batch_parameters = {'name', 'partition_id'}

property reader_options

property known_extensions

property reader_method

property base_directory

get_available_data_asset_names()
Return the list of asset names known by this generator.

Returns A list of available names

get_available_partition_ids(*generator_asset*)
Applies the current _partitioner to the batches available on generator_asset and returns a list of valid partition_id strings that can be used to identify batches of data.

Parameters **generator_asset** – the generator asset whose partitions should be returned.

Returns A list of partition_id strings

11.5.5 GlobReaderBatchKwargsGenerator

class great_expectations.datasource.generator.glob_reader_generator.**GlobReaderBatchKwargsGe**

Bases: *great_expectations.datasource.generator.batch_kwargs_generator.BatchKwargsGenerator*

GlobReaderBatchKwargsGenerator processes files in a directory according to glob patterns to produce batches of data.

A more interesting asset_glob might look like the following:

```
daily_logs:
  glob: daily_logs/*.csv
  partition_regex: daily_logs/((19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-
↪9]|[12][0-9]|3[01]))_(.*)\.csv
```

The "glob" key ensures that every csv file in the daily_logs directory is considered a batch for this data asset. The "partition_regex" key ensures that files whose basename begins with a date (with components hyphen, space, forward slash, period, or null separated) will be identified by a partition_id equal to just the date portion of their name.

A fully configured GlobReaderBatchKwargsGenerator in yml might look like the following:

```
my_datasource:
  class_name: PandasDatasource
  generators:
    my_generator:
      class_name: GlobReaderBatchKwargsGenerator
      base_directory: /var/log
      reader_options:
        sep: %
        header: 0
      reader_method: csv
      asset_globs:
        wifi_logs:
          glob: wifi*.log
```

(continues on next page)

(continued from previous page)

```
        partition_regex: wifi-((0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])-20\d\d).
↪*\.log
        reader_method: csv
```

recognized_batch_parameters = {'limit', 'name', 'reader_method', 'reader_options'}

property reader_options

property asset_globs

property reader_method

property base_directory

get_available_data_asset_names()
Return the list of asset names known by this generator.

Returns A list of available names

get_available_partition_ids(*generator_asset*)
Applies the current _partitioner to the batches available on generator_asset and returns a list of valid partition_id strings that can be used to identify batches of data.

Parameters **generator_asset** – the generator asset whose partitions should be returned.

Returns A list of partition_id strings

11.5.6 S3GlobReaderBatchKwargsGenerator

class great_expectations.datasource.generator.s3_generator.**S3GlobReaderBatchKwargsGenerator**

Bases: *great_expectations.datasource.generator.batch_kwargs_generator.BatchKwargsGenerator*

S3 Generator provides support for generating batches of data from an S3 bucket. For the S3 generator, assets must be individually defined using a prefix and glob, although several additional configuration parameters are available for assets (see below).

Example configuration:

```
datasources:
  my_datasource:
    ...
    generators:
      my_s3_generator:
        class_name: S3GlobReaderBatchKwargsGenerator
        bucket: my_bucket.my_organization.priv
```

(continues on next page)

(continued from previous page)

```
        reader_method: parquet  # This will be automatically inferred from suffix␣
↪where possible, but can be explicitly specified as well
        reader_options:  # Note that reader options can be specified globally or␣
↪per-asset
          sep: ","
        delimiter: "/"  # Note that this is the delimiter for the BUCKET KEYS. By␣
↪default it is "/"
        boto3_options:
          endpoint_url: $S3_ENDPOINT  # Use the S3_ENDPOINT environment variable␣
↪to determine which endpoint to use
        max_keys: 100  # The maximum number of keys to fetch in a single list_
↪objects request to s3. When accessing batch_kwargs through an iterator, the␣
↪iterator will silently refetch if more keys were available
        assets:
          my_first_asset:
            prefix: my_first_asset/
            regex_filter: .*  # The regex filter will filter the results returned␣
↪by S3 for the key and prefix to only those matching the regex
            dictionary_assets: True
          access_logs:
            prefix: access_logs
            regex_filter: access_logs/2019.*\.csv.gz
            sep: "~"
            max_keys: 100
```

property reader_options

property assets

property bucket

get_available_data_asset_names()
Return the list of asset names known by this generator.

Returns A list of available names

build_batch_kwargs_from_partition_id(*generator_asset*, *partition_id=None*, *reader_options=None*, *limit=None*)

get_available_partition_ids(*generator_asset*)
Applies the current _partitioner to the batches available on generator_asset and returns a list of valid partition_id strings that can be used to identify batches of data.

Parameters **generator_asset** – the generator asset whose partitions should be returned.

Returns A list of partition_id strings

11.5.7 DatabricksTableBatchKwargsGenerator

class great_expectations.datasource.generator.databricks_generator.**DatabricksTableBatchKwa**

Bases: *great_expectations.datasource.generator.batch_kwargs_generator.BatchKwargsGenerator*

Meant to be used in a Databricks notebook

get_available_data_asset_names()
Return the list of asset names known by this generator.

Returns A list of available names

11.6 Profile Module

class great_expectations.profile.base.**DataAssetProfiler**
Bases: object

classmethod validate(*data_asset*)

class great_expectations.profile.base.**DatasetProfiler**
Bases: *great_expectations.profile.base.DataAssetProfiler*

classmethod validate(*dataset*)

classmethod add_expectation_meta(*expectation*)

classmethod add_meta(*expectation_suite*, *batch_kwargs=None*)

classmethod profile(*data_asset*, *run_id=None*)

class great_expectations.profile.basic_dataset_profiler.**BasicDatasetProfilerBase**
Bases: *great_expectations.profile.base.DatasetProfiler*

BasicDatasetProfilerBase provides basic logic of inferring the type and the cardinality of columns that is used by the dataset profiler classes that extend this class.

INT_TYPE_NAMES = {'BIGINT', 'BYTEINT', 'DECIMAL', 'INT', 'INTEGER', 'IntegerType', 'Lo

FLOAT_TYPE_NAMES = {'DOUBLE_PRECISION', 'DoubleType', 'FLOAT', 'FLOAT4', 'FLOAT8', 'Fl

STRING_TYPE_NAMES = {'CHAR', 'STRING', 'StringType', 'TEXT', 'VARCHAR', 'str', 'string

BOOLEAN_TYPE_NAMES = {'BOOL', 'BOOLEAN', 'BooleanType', 'bool'}

DATETIME_TYPE_NAMES = {'DATE', 'DATETIME', 'DateType', 'TIME', 'TIMESTAMP', 'Timestamp

class great_expectations.profile.basic_dataset_profiler.**BasicDatasetProfiler**
Bases: *great_expectations.profile.basic_dataset_profiler.BasicDatasetProfilerBase*

BasicDatasetProfiler is inspired by the beloved pandas_profiling project.

The profiler examines a batch of data and creates a report that answers the basic questions most data practitioners would ask about a dataset during exploratory data analysis. The profiler reports how unique the values in the column are, as well as the percentage of empty values in it. Based on the column's type it provides a description of the column by computing a number of statistics, such as min, max, mean and median, for numeric columns, and distribution of values, when appropriate.

11.7 Render

11.7.1 Renderer Module

class great_expectations.render.renderer.renderer.**Renderer**

Bases: object

classmethod render(*ge_object*)

Site Builder

class great_expectations.render.renderer.site_builder.**SiteBuilder**(*data_context*, *store_backend*, *site_name=None*, *site_index_builder=None*, *show_how_to_buttons=True*, *site_section_builders=None*, *runtime_environment=None*, ***kwargs*)

Bases: object

SiteBuilder builds data documentation for the project defined by a DataContext.

A data documentation site consists of HTML pages for expectation suites, profiling and validation results, and an index.html page that links to all the pages.

The exact behavior of SiteBuilder is controlled by configuration in the DataContext's great_expectations.yml file.

Users can specify:

- which datasources to document (by default, all)
- whether to include expectations, validations and profiling results sections (by default, all)
- where the expectations and validations should be read from (filesystem or S3)
- where the HTML files should be written (filesystem or S3)
- which renderer and view class should be used to render each section

Here is an example of a minimal configuration for a site:

```
local_site:
    class_name: SiteBuilder
    store_backend:
        class_name: TupleS3StoreBackend
        bucket: data_docs.my_company.com
        prefix: /data_docs/
```

A more verbose configuration can also control individual sections and override renderers, views, and stores:

```
local_site:
    class_name: SiteBuilder
    store_backend:
        class_name: TupleS3StoreBackend
        bucket: data_docs.my_company.com
        prefix: /data_docs/
```

(continues on next page)

(continued from previous page)

```
    site_index_builder:
        class_name: DefaultSiteIndexBuilder

    # Verbose version:
    # index_builder:
    #     module_name: great_expectations.render.builder
    #     class_name: DefaultSiteIndexBuilder
    #     renderer:
    #         module_name: great_expectations.render.renderer
    #         class_name: SiteIndexPageRenderer
    #     view:
    #         module_name: great_expectations.render.view
    #         class_name: DefaultJinjaIndexPageView

    site_section_builders:
        # Minimal specification
        expectations:
            class_name: DefaultSiteSectionBuilder
            source_store_name: expectation_store
        renderer:
            module_name: great_expectations.render.renderer
            class_name: ExpectationSuitePageRenderer

        # More verbose specification with optional arguments
        validations:
            module_name: great_expectations.data_context.render
            class_name: DefaultSiteSectionBuilder
            source_store_name: local_validation_store
            renderer:
                module_name: great_expectations.render.renderer
                class_name: SiteIndexPageRenderer
            view:
                module_name: great_expectations.render.view
                class_name: DefaultJinjaIndexPageView
```

build(*resource_identifiers=None*)

Parameters **resource_identifiers** – a list of resource identifiers (ExpectationSuiteIdentifier, ValidationResultIdentifier). If specified, rebuild HTML (or other views the data docs site renders) only for the resources in this list. This supports incremental build of data docs sites (e.g., when a new validation result is created) and avoids full rebuild.

Returns

get_resource_url(*resource_identifier=None*)

Return the URL of the HTML document that renders a resource (e.g., an expectation suite or a validation result).

Parameters **resource_identifier** – ExpectationSuiteIdentifier, ValidationResultIdentifier or any other type's identifier. The argument is optional - when not supplied, the method returns the URL of the index page.

Returns URL (string)

class great_expectations.render.renderer.site_builder.**DefaultSiteSectionBuilder**(*name*, *data_context*, *target_store*, *source_store_na* *custom_styles_dire* *show_how_to_b* *run_id_filter=N* *validation_results_lim* *renderer=None*, *view=None*, ***kwargs*)

Bases: object

build(*resource_identifiers=None*)

class great_expectations.render.renderer.site_builder.**DefaultSiteIndexBuilder**(*name*, *site_name*, *data_context*, *target_store*, *custom_styles_directo* *show_how_to_butt* *validation_results_limit=* *renderer=None*, *view=None*, ***kwargs*)

Bases: object

add_resource_info_to_index_links_dict(*index_links_dict*, *expectation_suite_name*, *section_name*, *batch_identifier=None*, *run_id=None*, *validation_success=None*)

get_calls_to_action()

build()

class great_expectations.render.renderer.site_builder.**CallToActionButton**(*title*, *link*)

Bases: object

Page Renderers

Page Renderer

class great_expectations.render.renderer.page_renderer.**ValidationResultsPageRenderer**(*column_*
Bases: *great_expectations.render.renderer.renderer.Renderer*

render(*validation_results*)

class great_expectations.render.renderer.page_renderer.**ExpectationSuitePageRenderer**(*column_s*
Bases: *great_expectations.render.renderer.renderer.Renderer*

render(*expectations*)

class great_expectations.render.renderer.page_renderer.**ProfilingResultsPageRenderer**(*overview_*
col-
umn_sect
Bases: *great_expectations.render.renderer.renderer.Renderer*

render(*validation_results*)

Site Index Page Renderer

class great_expectations.render.renderer.site_index_page_renderer.**SiteIndexPageRenderer**
Bases: *great_expectations.render.renderer.renderer.Renderer*

classmethod render(*index_links_dict*)

Section Renderers

Column Section Renderer

great_expectations.render.renderer.column_section_renderer.**convert_to_string_and_escape**(*var*

class great_expectations.render.renderer.column_section_renderer.**ColumnSectionRenderer**
Bases: *great_expectations.render.renderer.renderer.Renderer*

class great_expectations.render.renderer.column_section_renderer.**ProfilingResultsColumnSect**

Bases: *great_expectations.render.renderer.column_section_renderer.ColumnSectionRenderer*

render(*evrs*, *section_name=None*, *column_type=None*)

class great_expectations.render.renderer.column_section_renderer.**ValidationResultsColumnSe**
Bases: *great_expectations.render.renderer.column_section_renderer.ColumnSectionRenderer*

render(*validation_results*)

class great_expectations.render.renderer.column_section_renderer.**ExpectationSuiteColumnSect**
Bases: *great_expectations.render.renderer.column_section_renderer.ColumnSectionRenderer*

render(*expectations*)

Other Section Renderer

class great_expectations.render.renderer.other_section_renderer.**ProfilingResultsOverviewSec**
Bases: *great_expectations.render.renderer.renderer.Renderer*

classmethod render(*evrs*, *section_name=None*)

Slack Renderer

class great_expectations.render.renderer.slack_renderer.**SlackRenderer**
Bases: *great_expectations.render.renderer.renderer.Renderer*

render(*validation_result=None*)

Content Blocks

class great_expectations.render.renderer.content_block.content_block.**ContentBlockRenderer**
Bases: *great_expectations.render.renderer.renderer.Renderer*

classmethod validate_input(*render_object*)

classmethod render(*render_object*, ***kwargs*)

classmethod list_available_expectations()

class great_expectations.render.renderer.content_block.bullet_list_content_block.**Expectatio**
Bases: *great_expectations.render.renderer.content_block.expectation_string.ExpectationStringRenderer*

great_expectations.render.renderer.content_block.expectation_string.**substitute_none_for_mis**

Utility function to plug Nones in when optional parameters are not specified in expectation kwargs.

Example

Input: kwargs={"a":1, "b":2}, kwarg_list=["c", "d"]

Output: {"a":1, "b":2, "c": None, "d": None}

This is helpful for standardizing the input objects for rendering functions. The alternative is lots of awkward *if "some_param" not in kwargs or kwargs["some_param"] == None:* clauses in renderers.

class great_expectations.render.renderer.content_block.expectation_string.**ExpectationString**
Bases: *great_expectations.render.renderer.content_block.content_block.ContentBlockRenderer*

classmethod expect_column_to_exist(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_unique_value_count_to_be_between(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_be_between(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_pair_values_A_to_be_greater_than_B(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_pair_values_to_be_equal(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_table_columns_to_match_ordered_list(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_multicolumn_values_to_be_unique(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_table_column_count_to_equal(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_table_column_count_to_be_between(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_table_row_count_to_be_between(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_table_row_count_to_equal(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_distinct_values_to_be_in_set(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_not_be_null(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_be_null(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_be_of_type(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_be_in_type_list(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_be_in_set(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_not_be_in_set(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_proportion_of_unique_values_to_be_between(*expectation*, *styling=None*, *include_column_name=True*

classmethod expect_column_values_to_be_increasing(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_be_decreasing(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_value_lengths_to_be_between(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_value_lengths_to_equal(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_match_regex(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_not_match_regex(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_match_regex_list(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_not_match_regex_list(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_match_strftime_format(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_be_dateutil_parseable(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_be_json_parseable(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_match_json_schema(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_distinct_values_to_contain_set(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_distinct_values_to_equal_set(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_mean_to_be_between(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_median_to_be_between(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_stdev_to_be_between(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_max_to_be_between(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_min_to_be_between(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_sum_to_be_between(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_most_common_value_to_be_in_set(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_quantile_values_to_be_between(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_kl_divergence_to_be_less_than(*expectation*, *styling=None*, *include_column_name=True*)

classmethod expect_column_values_to_be_unique(*expectation*, *styling=None*, *include_column_name=True*)

class great_expectations.render.renderer.content_block.profiling_overview_table_content_blo
Bases: *great_expectations.render.renderer.content_block.content_block.ContentBlockRenderer*

classmethod render(*ge_object*, *header_row=[]*)
Each expectation method should return a list of rows

classmethod expect_column_values_to_not_match_regex(*ge_object*)

classmethod expect_column_unique_value_count_to_be_between(*ge_object*)

classmethod expect_column_proportion_of_unique_values_to_be_between(*ge_object*)

classmethod expect_column_max_to_be_between(*ge_object*)

classmethod expect_column_mean_to_be_between(*ge_object*)

classmethod expect_column_values_to_not_be_null(*ge_object*)

classmethod expect_column_values_to_be_null(*ge_object*)

class great_expectations.render.renderer.content_block.validation_results_table_content_blo
Bases: *great_expectations.render.renderer.content_block.expectation_string.ExpectationStringRenderer*

class great_expectations.render.renderer.content_block.exception_list_content_block.**Except**
Bases: *great_expectations.render.renderer.content_block.content_block.ContentBlockRenderer*

Render a bullet list of exception messages raised for provided EVRs

11.7.2 View Module

class great_expectations.render.view.view.**NoOpTemplate**
Bases: object

render(*document*)

class great_expectations.render.view.view.**PrettyPrintTemplate**
Bases: object

render(*document*, *indent=2*)

class great_expectations.render.view.view.**DefaultJinjaView**(*custom_styles_directory=None*)
Bases: object

Defines a method for converting a document to human-consumable form

- Font Awesome 5.10.1
- Bootstrap 4.3.1
- jQuery 3.2.1
- Vega 5.3.5
- Vega-Lite 3.2.1
- Vega-Embed 4.0.0

render(*document*, *template=None*, ***kwargs*)

render_content_block(*context*, *content_block*, *index=None*, *content_block_id=None*)

get_html_escaped_json_string_from_dict(*source_dict*)

render_styling(*styling*)
Adds styling information suitable for an html tag.

Example styling block:

```
styling = {
    "classes": ["alert", "alert-warning"],
    "attributes": {
        "role": "alert",
        "data-toggle": "popover",
    },
    "styles" : {
        "padding" : "10px",
        "border-radius" : "2px",
    }
}
```

The above block returns a string similar to:

```
'class="alert alert-warning" role="alert" data-toggle="popover" style=
↪"padding: 10px; border-radius: 2px"'
```

"classes", "attributes" and "styles" are all optional parameters. If they aren't present, they simply won't be rendered.

Other dictionary keys are also allowed and ignored.

render_styling_from_string_template(*template*)
This method is a thin wrapper use to call *render_styling* from within jinja templates.

generate_html_element_uuid(*prefix=None*)

render_markdown(*markdown*)

render_string_template(*template*)

class great_expectations.render.view.view.**DefaultJinjaPageView**(*custom_styles_directory=None*)
Bases: *great_expectations.render.view.view.DefaultJinjaView*

class great_expectations.render.view.view.**DefaultJinjaIndexPageView**(*custom_styles_directory=None*)
Bases: *great_expectations.render.view.view.DefaultJinjaPageView*

class great_expectations.render.view.view.**DefaultJinjaSectionView**(*custom_styles_directory=None*)
Bases: *great_expectations.render.view.view.DefaultJinjaView*

class great_expectations.render.view.view.**DefaultJinjaComponentView**(*custom_styles_directory=None*)
Bases: *great_expectations.render.view.view.DefaultJinjaView*

11.8 Store Module

class great_expectations.data_context.store.store.**Store**(*store_backend=None*, *runtime_environment=None*)

Bases: object

A store is responsible for reading and writing Great Expectations objects to appropriate backends. It provides a generic API that the DataContext can use independently of any particular ORM and backend.

An implementation of a store will generally need to define the following:

- serialize
- deserialize
- _key_class (class of expected key type)

All keys must have a to_tuple() method.

serialize(*key*, *value*)

key_to_tuple(*key*)

tuple_to_key(*tuple_*)

deserialize(*key*, *value*)

get(*key*)

set(*key*, *value*)

list_keys()

has_key(*key*)

class great_expectations.data_context.store.store_backend.**StoreBackend**(*fixed_length_key=False*)

Bases: object

IGNORED_FILES = ['.ipynb_checkpoints']

property fixed_length_key

get(*key*)

set(*key*, *value*, ***kwargs*)

has_key(*key*)

get_url_for_key(*key*, *protocol=None*)

abstract list_keys(*prefix=()*)

is_ignored_key(*key*)

class great_expectations.data_context.store.store_backend.**InMemoryStoreBackend**(*runtime_environ*
fixed_length_key

Bases: *great_expectations.data_context.store.store_backend.StoreBackend*

Uses an in-memory dictionary as a store backend.

list_keys(*prefix=()*)

11.9 Validation Operators Module

11.9.1 ValidationOperator

class great_expectations.validation_operators.**ValidationOperator**

Bases: object

The base class of all validation operators.

It defines the signature of the public run method - this is the only contract re operators' API. Everything else is up to the implementors of validation operator classes that will be the descendants of this base class.

run(*assets_to_validate*, *run_id*, *evaluation_parameters=None*)

11.9.2 ActionListValidationOperator

class great_expectations.validation_operators.**ActionListValidationOperator**(*data_context*, *action_list*)

Bases: great_expectations.validation_operators.validation_operators.ValidationOperator

ActionListValidationOperator is a validation operator that validates each batch in the list that is passed to its run method and then invokes a list of configured actions on every validation result.

A user can configure the list of actions to invoke.

Each action in the list must be an instance of ValidationAction class (or its descendants).

Below is an example of this operator's configuration:

```
action_list_operator:
    class_name: ActionListValidationOperator
    action_list:
      - name: store_validation_result
        action:
          class_name: StoreValidationResultAction
          target_store_name: validations_store
      - name: store_evaluation_params
        action:
          class_name: StoreEvaluationParametersAction
          target_store_name: evaluation_parameter_store
      - name: send_slack_notification_on_validation_result
        action:
          class_name: SlackNotificationAction
          # put the actual webhook URL in the uncommitted/config_variables.yml␣
↪file
          slack_webhook: ${validation_notification_slack_webhook}
         notify_on: all # possible values: "all", "failure", "success"
          renderer:
            module_name: great_expectations.render.renderer.slack_renderer
            class_name: SlackRenderer
```

run(*assets_to_validate*, *run_id*, *evaluation_parameters=None*)

11.9.3 WarningAndFailureExpectationSuitesValidationOperator

class great_expectations.validation_operators.**WarningAndFailureExpectationSuitesValidationO**

Bases: great_expectations.validation_operators.validation_operators.ActionListValidationOperator

WarningAndFailureExpectationSuitesValidationOperator is a validation operator that accepts a list batches of data assets (or the information necessary to fetch these batches). The operator retrieves 2 expectation suites for each data asset/batch - one containing the critical expectations ("failure") and the other containing non-critical expectations ("warning"). By default, the operator assumes that the first is called "failure" and the second is called "warning", but "base_expectation_suite_name" attribute can be specified in the operator's configuration to make sure it searched for "{base_expectation_suite_name}.failure" and {base_expectation_suite_name}.warning" expectation suites for each data asset.

The operator validates each batch against its "failure" and "warning" expectation suites and invokes a list of actions on every validation result.

The list of these actions is specified in the operator's configuration

Each action in the list must be an instance of ValidationAction class (or its descendants).

The operator sends a Slack notification (if "slack_webhook" is present in its config). The "notify_on" config property controls whether the notification should be sent only in the case of failure ("failure"), only in the case of success ("success"), or always ("all").

Below is an example of this operator's configuration:

```
run_warning_and_failure_expectation_suites:
    class_name: WarningAndFailureExpectationSuitesValidationOperator
    # put the actual webhook URL in the uncommitted/config_variables.yml file
    slack_webhook: ${validation_notification_slack_webhook}
    action_list:
      - name: store_validation_result
        action:
          class_name: StoreValidationResultAction
          target_store_name: validations_store
      - name: store_evaluation_params
        action:
          class_name: StoreEvaluationParametersAction
          target_store_name: evaluation_parameter_store
```

The operator returns an object that looks like the example below.

The value of "success" is True if no critical expectation suites ("failure") failed to validate (non-critial ("warning") expectation suites are allowed to fail without affecting the success status of the run:

```
{
    "batch_identifiers": [list, of, batch, identifiers],
    "success": True/False,
    "failure": {
        "expectation_suite_identifier": {
            "validation_result": validation_result,
            "action_results": {
                "action name": "action result object"
            }
        }
    },
    "warning": {
        "expectation_suite_identifier": {
            "validation_result": validation_result,
            "action_results": {
                "action name": "action result object"
            }
        }
    }
}
```

run(*assets_to_validate*, *run_id*, *base_expectation_suite_name=None*, *evaluation_parameters=None*)

11.10 Great Expectations Module

class great_expectations.types.**ClassConfig**(*class_name*, *module_name=None*)

Defines information sufficient to identify a class to be (dynamically) loaded for a DataContext.

property class_name

property module_name

CHAPTER

TWELVE

INDEX

- genindex
- modindex

PYTHON MODULE INDEX

g

INDEX

A

B

C

D

E

F

G

H

I

K

S

T

U

V

W

Y