Suffyan Asad Follow

Dec 18 · 10 min read · Listen

Save

PySpark — Writing Unit Tests for Spark SQL Transformations

An introduction to writing unit tests for Spark SQL using Python `unittest` library, covering creating test data of types including Map, List and Struct, and validating the results.

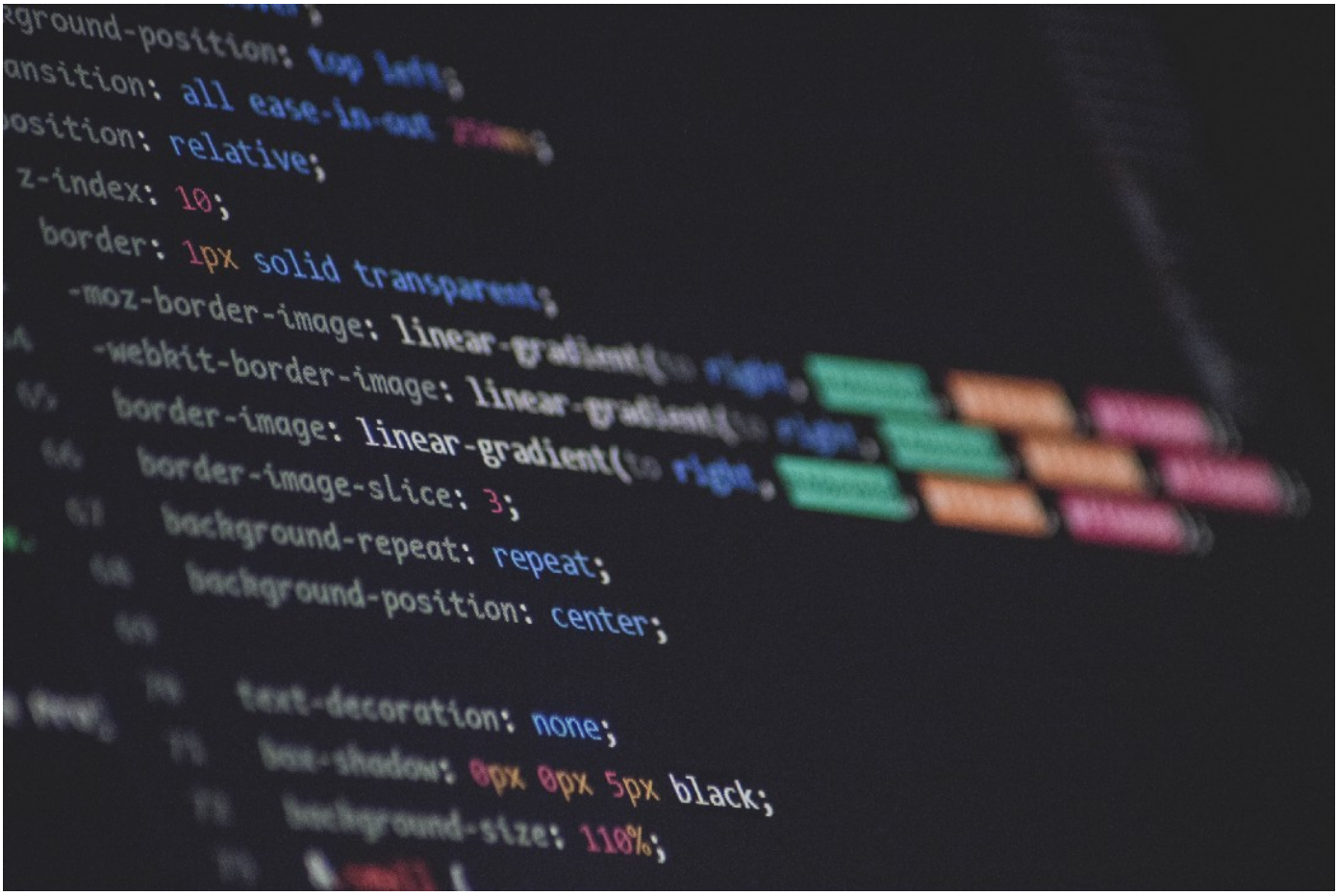


Photo by Maik Jonietz on Unsplash

Introduction

Unit testing is one of the most widely used ways of testing code. Unit tests are supposed to test the basic components, or "units" of code by running them for test inputs and comparing the outputs with the expected outputs. Unit testing spark transformations and operations is not that different, and here the units are the Spark SQL operations on the data frames.

Most of the calculations in Spark jobs are written in the form of transformations, that take one primary data frame, and other inputs, including other data frames, and add/modify/remove columns while performing calculations. These transformations are linked in series of steps that execute one after the other, and perform the calculations on the data as needed. They start with steps that load the data from external sources, and end with steps that output data to the external sources.

This article is an introduction to writing unit tests for these Spark SQL transformations. The focus is on creating test data, including data types that support complex and nested data — List, Map and Struct types, and on validating the results of transformations.

The article presents writing tests for PySpark SQL transformations on DataFrames, and do not include testing RDD operations.

Unit tests using `unittest` framework

This article presents writing unit tests for Spark transformations in Python (PySpark) using the `unittest` framework. The `unittest` framework in built into Python for unit testing. Presenting detailed introduction to the `unittest` framework, and comparison with other frameworks such as `PyTest` is beyond the scope of this article. For continuation, a basic unit-test and relevant details are below.

`add` is a simple function that adds two numbers:

```
1  def add(x:int, y:int) -> int:
2      return x+y
```

simple_test.py hosted with ❤ by GitHub view raw

Addition function to test

A unit test using the `unittest` framework will can be written as:

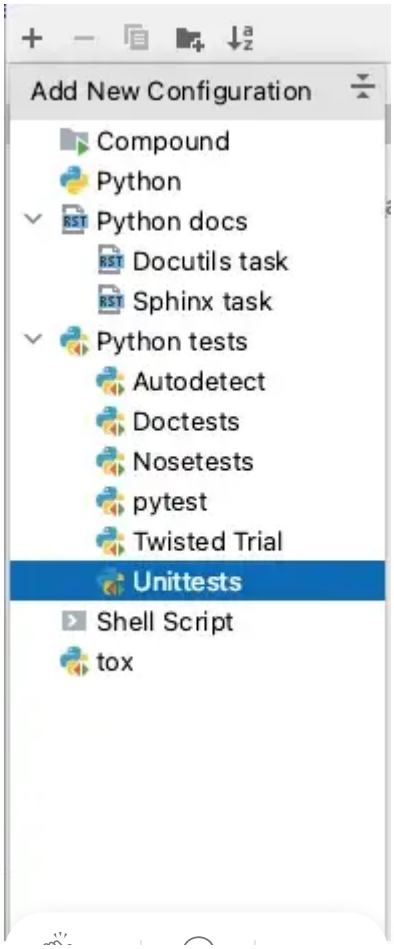

```
4
5  class TestAddition(unittest.TestCase):
6      def test_add(self):
7          assert add(1, 2) == 3
```

simple_test_.py hosted with ❤ by GitHub view raw

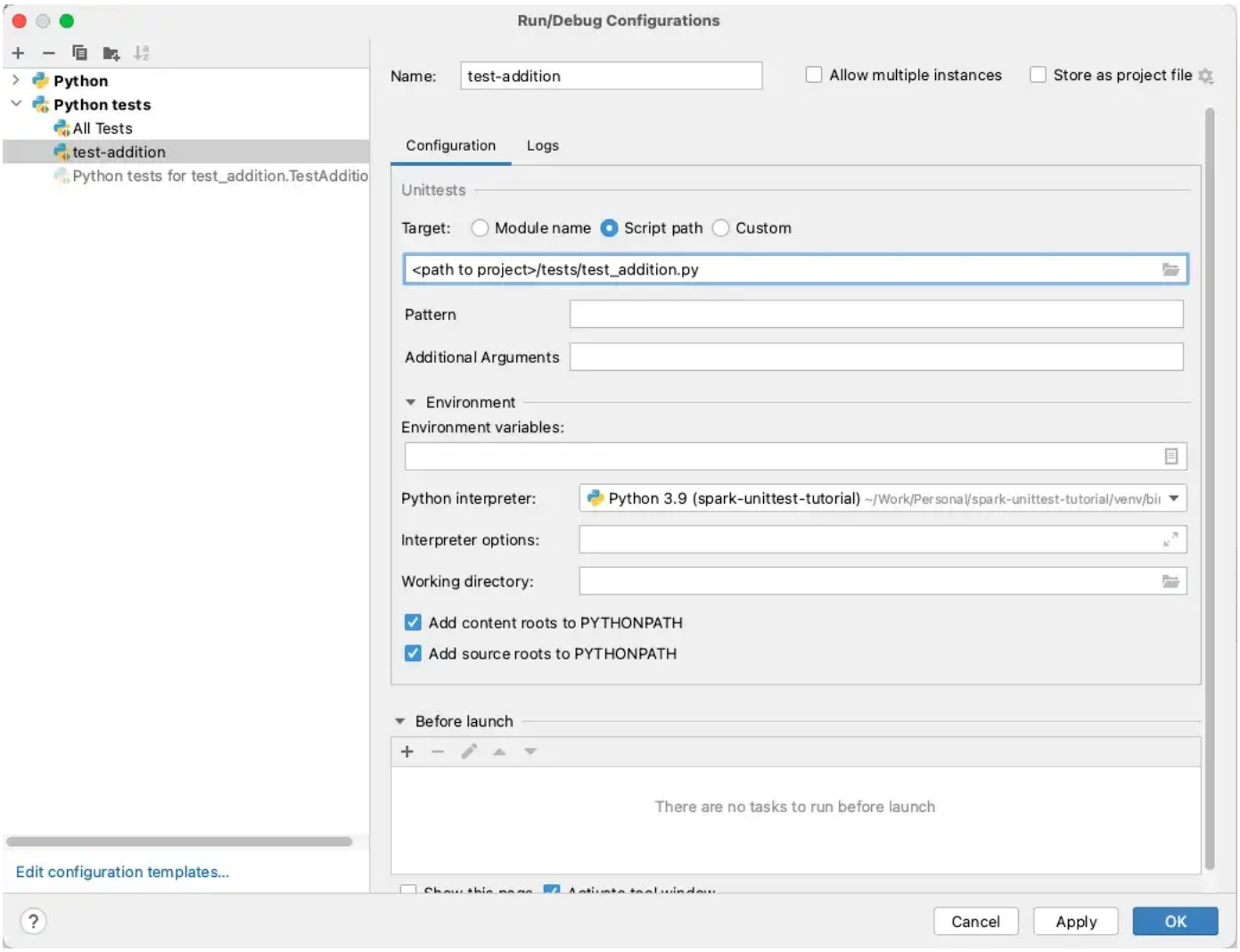
Unit test to test the addition function

The class `TestAddition` is initiated by passing in `unittest.TestCase` class, which initiates this class as a unit-test class. The `test_add` function validates the addition operation. We'll name the file containing the test class as `test_addition.py` .

This test can be executed in the PyCharm IDE by creating a unit test Run/Debug run configuration for the test class created above:

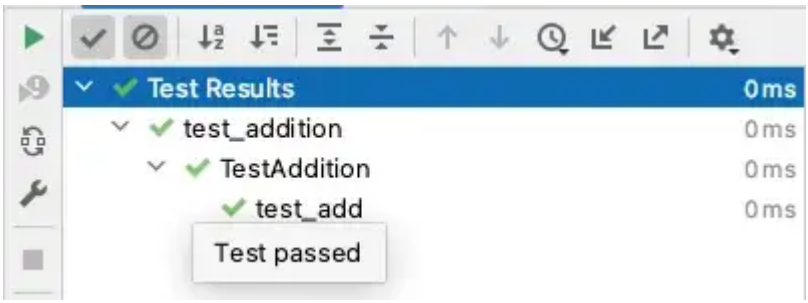


Create 7 tests



Configure the unit tests run configuration parameters

The tests can now be executed by running the run configuration:



Execution of unit tests

Further details and tutorials about using the unittest framework to write tests can be found in the official documentation:

unittest - Unit testing framework

Source code: Lib/unittest/__init__.py (If you are already familiar with the basic concepts of testing, you might want...

docs.python.org

Writing tests using pytest

In Python, `unittest` is not the only framework for writing tests, and `pytest` is a popular alternative. The focus of this article is on writing unit tests for Spark, and the concepts should translate to `pytest` or any other unit test framework.

Link to pytest documentation for reference:

pytest: helps you write better programs - pytest documentation

The maintainers of pytest and thousands of other packages are working with Tidelift to deliver commercial support and...

docs.pytest.org

Code repository

The code examples used in the article can be downloaded from:

GitHub - SA01/spark-unittest-tutorial

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or...

github.com

Initializing Spark Session for testing

The first step for testing Spark operations and transformation is to create a Spark session (`pyspark.sql.SparkSession`). In order to run unit-tests from the command line or from within the IDE, the best and quick option is to configure Spark to run in local mode. Since this Spark session will be needed by all the tests in the test suite, it needs

to be created before the tests start. The `unittest` framework provides a function `setUp` (Reference: https://docs.python.org/3/library/unittest.html#unittest.TestCase.setUp) that runs immediately before running each test.

This code for setting-up the Spark Session itself passes `local` as master, followed by the required number of cores. Please refer to the following link to Spark documentation describing various modes and the ways to configure them:

Submitting Applications

The spark-submit script in Spark's bin directory is used to launch applications on a cluster. It can use all of Spark's...

spark.apache.org

Following code sets up the Spark session for test classes, using 4 cores on the local machine and 8GB RAM for the driver:

Creating test class and setting up Spark session in the setUp function

Alternatively, Spark session can be created in each test if they require setting-up the session differently.

Writing the unit tests

Each unit test performs the following steps:

- Create a data frame containing test data to pass to the function being tested.
- Run the transformation and get the results.
- Validate the existence of expected column(s) in the result data frame added by the function.
- Validate the data in the result column(s).

In addition to these major steps, other steps can be added as needed.

After setting-up the Spark session, the next step is to create test data, which, in case of Spark, is data frames (instances of `pyspark.sql.DataFrame`). These data-frames will be run through the data-frame operations in unit tests.

Unit tests with simple data

Test data frames can be created by creating each row of the data as a tuple, and the entire data as the list of those tuples. For example, the following function takes a data-frame, and adds two columns called `first` and `second`. The result is stored in a column called `sum`:

Function to add two columns and save the result in a third column named sum

The unit-test can be written as:

Unit test to create sample data, and validate the results after running the add_columns function

In the above example, values in each tuple in the `test_data` are mapped to columns `first` and `second`. After running the transformation, the column `sum` is extracted from the result using the `collect` method, and is validated against a list of expected

values using the `assertListEqual` function. Also, addition of the column `sum` is validated by confirming its presence in the result data frame.

This was a very simple example with 2 columns, both integer. Creating other basic data types, e.g. string, date/time and float data is also simple and can be done using tuples. Types such as date (`pyspark.sql.types.DateType`) or timestamp (`pyspark.sql.types.TimstampType`) can either be created as instances of `datetime.date` or `datetime.datetime` while creating the test data list of rows, or, can be loaded into the `DataFrame` as strings, and casted to appropriate types later.

As an example, the following function calculates running total of products sold by date of sale, and the unit test to create the test data, and run the test:

Function to create running total, and the unit test to validate it

Alternatively, the date column can be initially passed in the `createDataFrame` function as a string, and then converted to the appropriate data-type:

Alternate way of writing the test

Printing the schema and calling the `.show()` on the `DataFrame` gives the following output:

Output of the test

The `order_date` column is of type `pyspark.sql.types.DateType`. Also, the numeric values passed in the column `order_id` have been loaded as long, and may require casting them to integer in some cases.

Please note that decimal and float data (`pyspark.sql.types.FloatType`, `pyspark.sql.types.DecimalType`) may not match exactly in assertions, and may require using `assertAlmostEqual` with appropriate number of decimal places to match.

Creating and validating map column — custom schema

Creating map data for testing transformations is similar, it can be created by creating instances of python dictionary (`dict`) in the test data that is used to create the data frame:

Creating data with Map type column for the unit test

In this example, the values of the map column `sales` are initiated as dictionaries - instances of `dict` in each row of test data. Running the above code in a test gives the following output:

Output: Map column created.

The data has been loaded successfully, but there are two problems with it:

1. The numeric columns have type `long` whereas we wanted `integer`. Both `order_id` column as well as the value in the map column `sales` have long data-type.
2. The parameter `valueContainsNull` is true despite the fact that there is no null value for any key in any row of the map column `sales`.

These problems can be fixed by passing a custom schema instead of a list of columns when calling the `createDataFrame` function. The above code can be modified as:

Creating Map column with schema

The schema defines the datatype of each column that will be applied when creating the test data frame. We get the following output upon running the updated code:

Fixed data types by providing schema when creating the data frame

Additionally, if a schema is provided, Spark throws exceptions if the data provided doesn't match the schema. For example, if the `order_id` column has string values in it, we get the `TypeError` exception:

Exception when the data and schema do not match

Additionally, please remember that the `MapType` in Spark supports only one data type for keys and one data type for values. In this example, the `sales` column cannot contain a row with a string value as it will not be a valid map. `TypeError` is thrown at the time of creating the data frame if this issue is present and schema has been provided. If a schema is not present, the invalid string value is read as `null`, or an exception is raised if one row contains map of a different type then the rest of the column.

For example, the following code will read the string value 3 as null:

Type mismatch in data — one row contains a string value in the map column

And the following code with one row of type `Map<string, string>` and all other rows containing values of type `Map<string, int>` will throw a `TypeError` exception with the message `TypeError: value of map field sales: Can not merge type <class 'pyspark.sql.types.LongType'> and <class 'pyspark.sql.types.StringType'>:`

Type mismatch in data — one row contains map with different data types

Creating and validating data with array column

Array column can be created by creating list in the data, and then loading that list into Spark as a data frame. Schema should be provided to tell Spark about the data types of elements of the arrays:

Creating data with array column for the unit test

This function creates a data frame with one integer, one string, two date and lastly, one array column named `sales`. Upon running the test, the following output is produced:

Output: Data with array column created

Array columns return a list of lists when they are pulled out of Spark using the `collect` function. The data type of the elements of each list is the equivalent Python data type of the types of the data frame schema. For example, a column containing list of `IntegerType` will return a list containing each element of type `List[int]`.

Validation can be done by using the `assertListEqual` function:

Validating the result in array column

If the array column contains dates (`pyspark.sql.types.DateType`), the values that are returned on collecting the data frame using are of type `[datetime.datetime.date](<http://datetime.datetime.date>)` as we have already seen in a previous example. Similarly, the column contains an array of dates, the data returned from Spark is a list, and each element of the result is a list of dates.

To explain this, consider an example where the source data contains product, date of sale, and mode of sale (online or retail), and number of sales. A transformation groups this data by product and mode of sale, and collects both date of sales, and number of sales into arrays by using the `collect_list` function.

The resultant data can be validated in two ways:

1. By comparing the list of dates with an expected list of dates containing instances of `[datetime.datetime.date](<http://datetime.datetime.date>)`.
2. Alternatively, the dates can be converted to string, either within Spark by calling the `pyspark.sql.functions.date_format` function, or by calling `strptime` function on each element of the date lists returned by Spark.

The following code uses the first method to validate:

Creating test data with array column containing dates

Following output is received when the test is run:

Output of unit test — validation of result containing array columns

Creating and validating struct column

Creating Struct data is a little different because there is no direct mapping between Python data types and the struct data type in Spark. Due to this, both creating and

validating test data is a little different.

One way to create a struct column is to create test data as python dictionary, keeping all key names consistent. Then, instead of defining the column as a map column, define it as a struct column in the data frame schema. The fields defined in the struct column definition should have the same names as the keys in the dictionary. Any key that is not present in a row will appear as null for that row in the data frame. Any key that has not been defined in the struct will be ignored.

This has been demonstrated in the following code example:

Creating test data containing struct column

And it gives the following output:

Created test data containing struct column

For validation, more work is required after collecting the data frame, because the struct column elements are received as nested instances of `spark.sql.types.Row`. Therefore validating the results in the unit test can be done in following ways:

1. Map the elements of the results to other types e.g. python dictionary, or list. This can get complicated if the data
2. Select the sub-columns of the struct as separate columns, and then collect the `DataFrame`.
3. Convert the struct column to json using the `pyspark.sql.functions.to_json` function, and then collect to receive python dictionary that can be validated in the test.

Following is the code implementation of validation of the data created above using all three methods:

Techniques to validate data in struct column

Testing with data files (CSV or JSON)

In addition to providing data from the code within the function, another way is to load data from a file. Data can be loaded using Spark's built-in `DataFrameReader` methods such as `.parquet()`, `.csv()` or `.json()`. It is better to use CSV or JSON data because they are human-readable formats, and the data in them can be created or updated manually.

This method is better when testing on large data, specially if multiple transformations are required to be tested on relatively large data, and if the expected result is also large and is present as a file. Also, this technique is useful if test data is not manually created, and is an output of a process. Saving the data in CSV or JSON makes it easily to modify manually. Providing a schema is also recommended when using CVS or json files to ensure that the data frame schema and data types match the schema and data types of the data that will be received by the function being tested in production runs.

For example, using the same function `group_sales_by_type` but loading data from the following CSV file:

CSV file containing test data

And the expected result data:

CSV file containing the expected result of the transformation

The test will be written as:

Unit test that reads test data and expected data from CSV files

And it should give the expected results.

I have tried to cover writing unit tests for Spark transformations focusing on creating test data that can be passed in Spark SQL transformations, and validating the output data. Please share your feedback in comments.

Spark Pyspark Unit Testing Python Data Engineering