Published in Towards Data Science

Khuyen Tran Follow
Mar 26, 2022 · 5 min read · · Listen

Save

Validate Your pandas DataFrame with Pandera

Make Sure Your Data Matches Your Expectation

Motivation

In a data science project, it is not only important to test your functions, but it is also important to test your data to make sure they work as you expected.

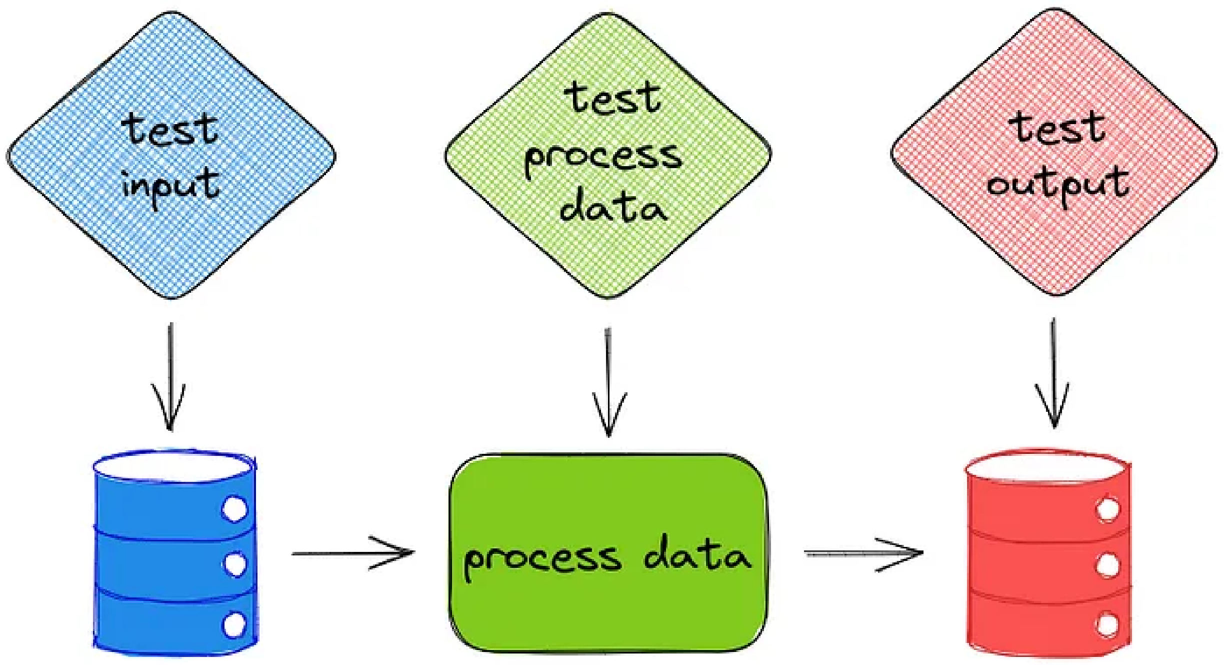


Image by Author

In the previous article, I showed how to use Great Expectations to validate your data.

Great Expectations: Always Know What to Expect From Your Data
Ensure Your Data Works as Expected Using Python
towardsdatascience.com

Even though Great Expectations provide a lot of useful utilities, it can be complicated to create a validation suite with Great Expectations. For a small data science project, using Great Expectations can be overkill.

That is why in this article we will learn about Pandera, a simple Python library for validating a pandas DataFrame.

To install Pandera, type:

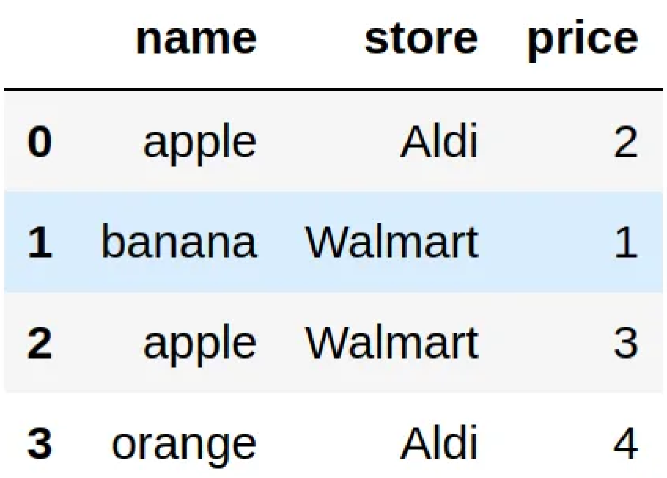
```
pip install pandera
```

Introduction

To learn how Pandera works, let's start with creating a simple dataset:

```python
import pandas as pd

fruits = pd.DataFrame(
    {
        "name": ["apple", "banana", "apple", "orange"],
        "store": ["Aldi", "Walmart", "Walmart", "Aldi"],
        "price": [2, 1, 3, 4],
    }
)

fruits
```

dataset.py hosted with ❤ by GitHub view raw



	name	store	price
0	apple	Aldi	2
1	banana	Walmart	1
2	apple	Walmart	3
3	orange	Aldi	4

Image by Author

Imagine this scenario. Your manager told you that there can only be certain fruits and stores in the dataset, and the price must be less than 4.

```python
available_fruits = ["apple", "banana", "orange"]
nearby_stores = ["Aldi", "Walmart"]
```

validate.py hosted with ❤ by GitHub view raw

To make sure your data follow these conditions, checking your data manually can cost too much time, especially when your data is big. Is there a way that you can automate this process?

That is when Pandera comes in handy. Specifically, we:

- Create multiple tests for the entire dataset using `DataFrameSchema`
- Create multiple tests for each column using `Column`

- Specify the type of test using `Check`

```python
import pandera as pa
from pandera import Column, Check

schema = pa.DataFrameSchema(
    {
        "name": Column(str, Check.isin(available_fruits)),
        "store": Column(str, Check.isin(nearby_stores)),
        "price": Column(int, Check.less_than(4)),
    }
)
schema.validate(fruits)
```

validate.py hosted with ❤ by GitHub view raw

```
SchemaError: <Schema Column(name=price, type=DataType(int64))> failed element-wise validator
0:
<Check less_than: less_than(4)>
failure cases:
   index  failure_case
0      3             4
```

In the code above:

- `"name": Column(str, Check.isin(available_fruits))` checks if the column `name` is of type string and if all values of the column `name` are inside a specified list.
- `"price": Column(int, Check.less_than(4))` checks if all values in the column `price` are of type `int` and are less than 4.
- Since not all values in the column `price` are less than 4, the test fails.

Find other built-in `Checks` methods here.

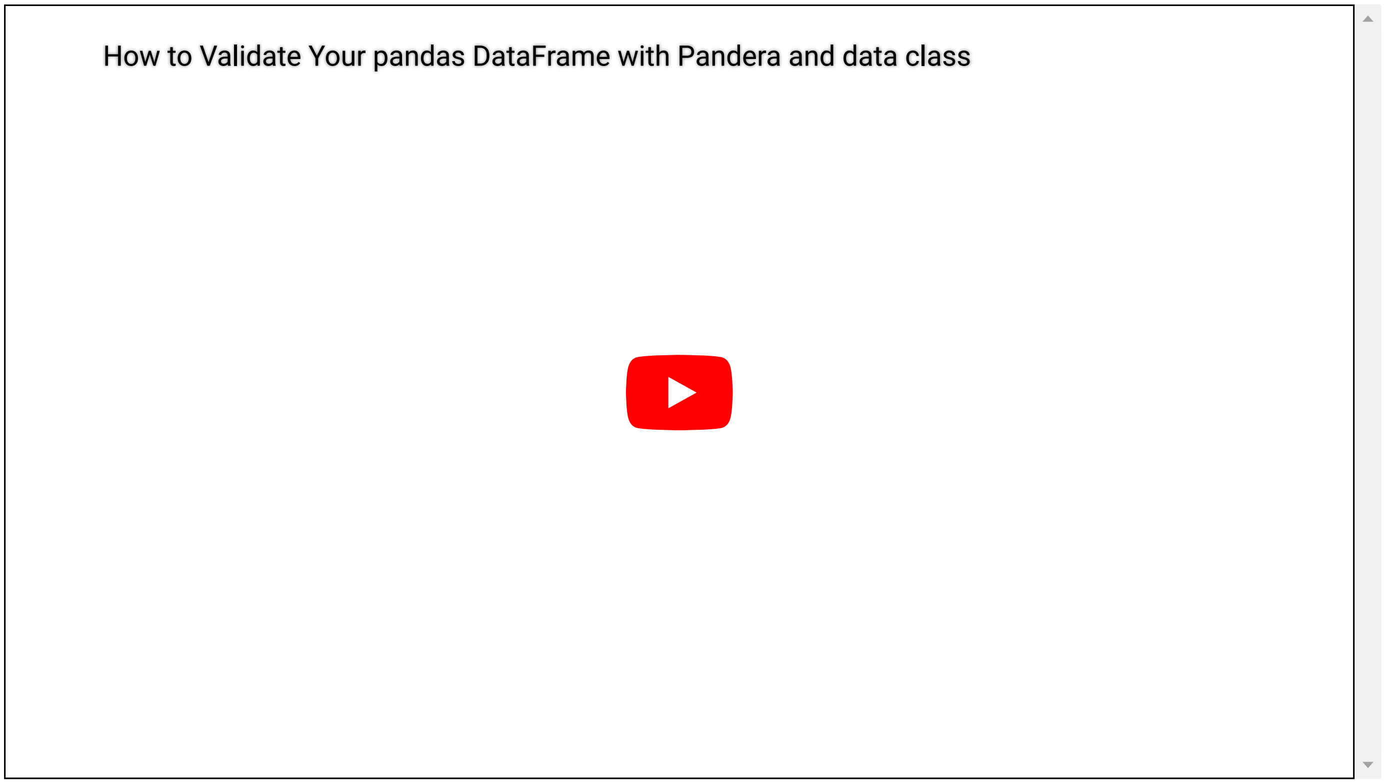
Custom Checks

We can also create custom checks using `lambda` . In the code below, `Check(lambda price: sum(price) < 20)` checks if the sum of the column `price` is less than 20.

```python
schema = pa.DataFrameSchema(
    {
        "name": Column(str, Check.isin(available_fruits)),
        "store": Column(str, Check.isin(nearby_stores)),
        "price": Column(
            int, [Check.less_than(5), Check(lambda price: sum(price) < 20)]
        ),
    }
)
schema.validate(fruits)
```

custom_checks.py hosted with ❤ by GitHub view raw

Schema Model

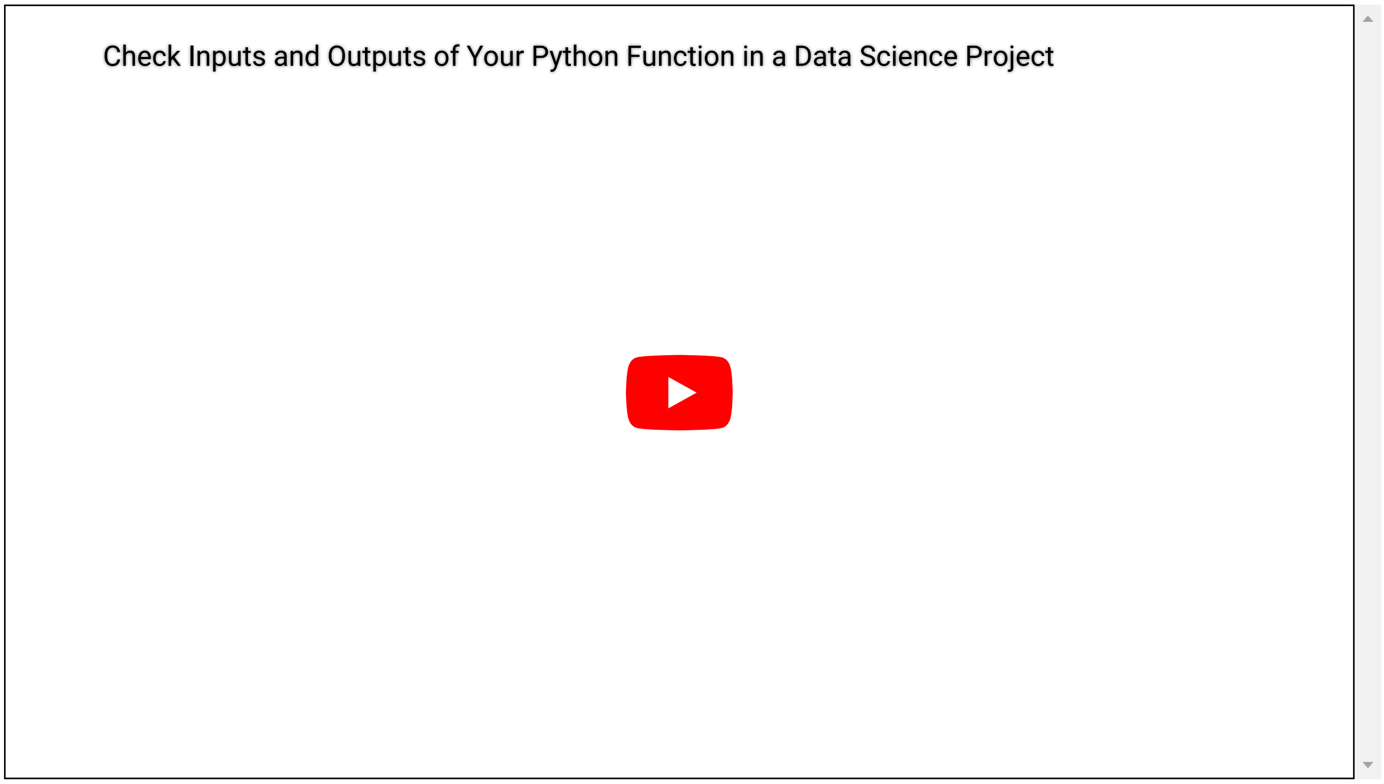

How to Validate Your pandas DataFrame with Pandera and data class

When our tests are complicated, using dataclass can make our tests look much cleaner than using a dictionary. Luckily, Pandera also allows us to create tests using a dataclass instead of a dictionary.

```python
from pandera.typing import Series

class Schema(pa.SchemaModel):
    name: Series[str] = pa.Field(isin=available_fruits)
    store: Series[str] = pa.Field(isin=nearby_stores)
    price: Series[int] = pa.Field(le=5)

    @pa.check("price")
    def price_sum_lt_20(cls, price: Series[int]) -> Series[bool]:
        return sum(price) < 20

Schema.validate(fruits)
```

dataclass.py hosted with ❤ by GitHub view raw

Validation Decorator


Check Inputs and Outputs of Your Python Function in a Data Science Project

Check Input

Now that we know how to create tests for our data, how do we use it to test the input of our function? A straightforward approach is to add `schema.validate(input)` inside a function.

```python
fruits = pd.DataFrame(
    {
        "name": ["apple", "banana", "apple", "orange"],
        "store": ["Aldi", "Walmart", "Walmart", "Aldi"],
        "price": [2, 1, 3, 4],
    }
)

schema = pa.DataFrameSchema(
    {
        "name": Column(str, Check.isin(available_fruits)),
        "store": Column(str, Check.isin(nearby_stores)),
        "price": Column(int, Check.less_than(5)),
    }
)


def get_total_price(fruits: pd.DataFrame, schema: pa.DataFrameSchema):
    validated = schema.validate(fruits)
    return validated["price"].sum()


get_total_price(fruits, schema)
```

pipeline.py hosted with ❤ by GitHub view raw

However, this approach makes it difficult for us to test our function. Since the argument of `get_total_price` is both `fruits` and `schema`, we need to include both of these arguments inside the test:

```python
def test_get_total_price():
    fruits = pd.DataFrame({'name': ['apple', 'banana'], 'store': ['Aldi', 'Walmart'], 'price': [1, 2]})

    # Need to include schema in the unit test
    schema = pa.DataFrameSchema(
        {
            "name": Column(str, Check.isin(available_fruits)),
            "store": Column(str, Check.isin(nearby_stores)),
            "price": Column(int, Check.less_than(5)),
        }
    )
    assert get_total_price(fruits, schema) == 3
```

test_get_total_price.py hosted with ❤ by GitHub view raw

`test_get_total_price` tests both the data and the function. Because a unit test should only test one thing, including data validation inside a function is not ideal.

Pandera provides a solution for this with the `check_input` decorator. The argument of this decorator is used to validate the input of the function.

```python
from pandera import check_input

@check_input(schema)
def get_total_price(fruits: pd.DataFrame):
    return fruits.price.sum()

get_total_price(fruits)
```

check_input.py hosted with ❤ by GitHub view raw

If the input is not valid, Pandera will raise an error before the input is processed by your function:

```python
fruits = pd.DataFrame(
    {
        "name": ["apple", "banana", "apple", "orange"],
        "store": ["Aldi", "Walmart", "Walmart", "Aldi"],
        "price": ["2", "1", "3", "4"],
    }
)

@check_input(schema)
def get_total_price(fruits: pd.DataFrame):
    return fruits.price.sum()

get_total_price(fruits)
```

validate.py hosted with ❤ by GitHub view raw

```
SchemaError: error in check_input decorator of function 'get_total_price': expected series
'price' to have type int64, got object
```

Validating data before processing is very nice since it **prevents** us from **wasting a significant amount of time on processing the data.**

Check Output

We can also use Pandera's `check_output` decorator to check the output of a function:

```python
from pandera import check_output

fruits_nearby = pd.DataFrame(
    {
        "name": ["apple", "banana", "apple", "orange"],
        "store": ["Aldi", "Walmart", "Walmart", "Aldi"],
        "price": [2, 1, 3, 4],
    }
)

fruits_faraway = pd.DataFrame(
    {
        "name": ["apple", "banana", "apple", "orange"],
        "store": ["Whole Foods", "Whole Foods", "Schnucks", "Schnucks"],
        "price": [3, 2, 4, 5],
    }
)

out_schema = pa.DataFrameSchema(
    {"store": Column(str, Check.isin(["Aldi", "Walmart", "Whole Foods", "Schnucks"]))}
)


@check_output(out_schema)
def combine_fruits(fruits_nearby: pd.DataFrame, fruits_faraway: pd.DataFrame):
    fruits = pd.concat([fruits_nearby, fruits_faraway])
    return fruits


combine_fruits(fruits_nearby, fruits_faraway)
```

check_output.py hosted with ❤ by GitHub view raw

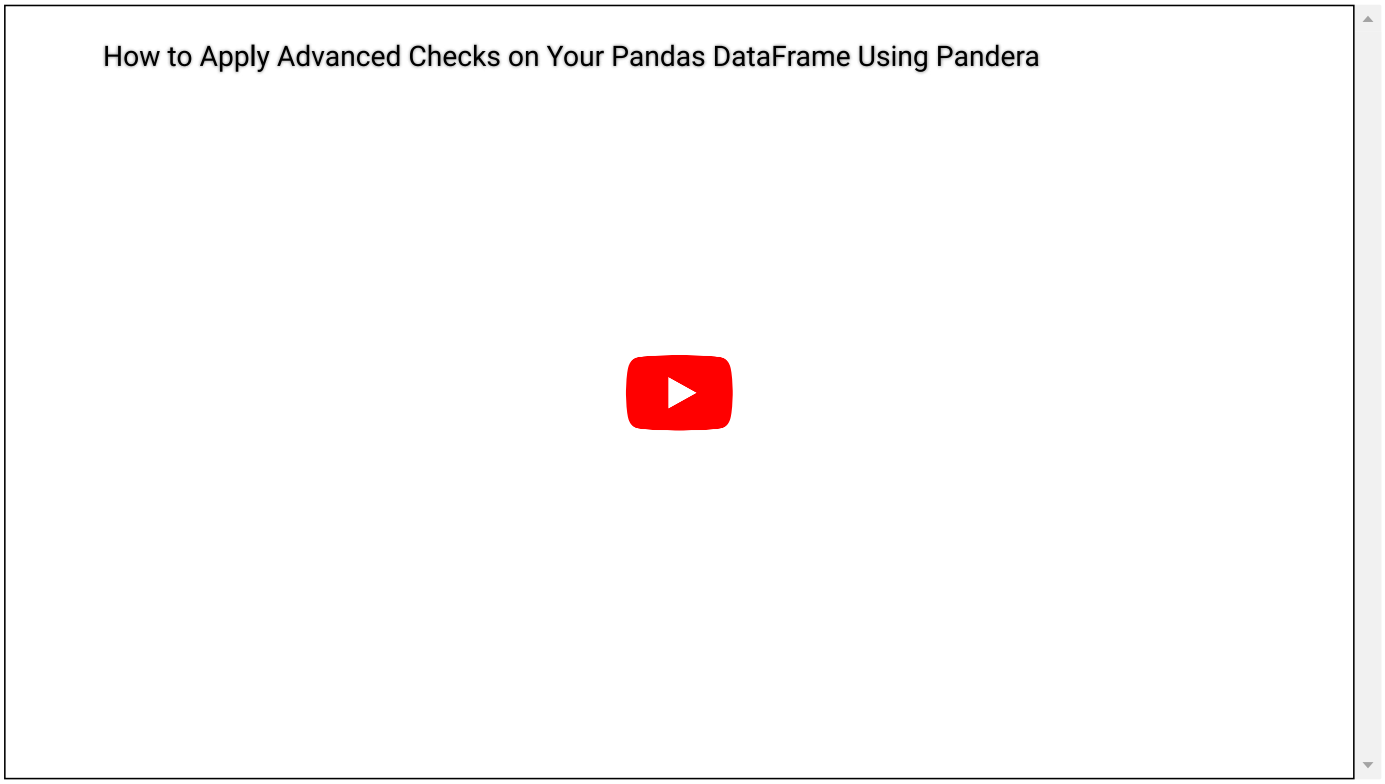
Check Both Inputs and Outputs

Now you might wonder, is there a way to check both inputs and outputs? We can do that using the decorator `check_io` :

```python
from pandera import check_io

in_schema = pa.DataFrameSchema({"store": Column(str)})

out_schema = pa.DataFrameSchema(
    {"store": Column(str, Check.isin(["Aldi", "Walmart", "Whole Foods", "Schnucks"]))}
)


@check_io(fruits_nearby=in_schema, fruits_faraway=in_schema, out=out_schema)
def combine_fruits(fruits_nearby: pd.DataFrame, fruits_faraway: pd.DataFrame):
    fruits = pd.concat([fruits_nearby, fruits_faraway])
    return fruits


combine_fruits(fruits_nearby, fruits_faraway)
```

check_both.py hosted with ❤ by GitHub view raw

Other Arguments for Column Validation


How to Apply Advanced Checks on Your Pandas DataFrame Using Pandera

541 | 6 | •••

Deal with Null Values

By default, Pandera will raise an error if there are null values in a column we are testing. If null values are acceptable, add `nullable=True` to our `Column` class:

```python
import numpy as np

fruits = fruits = pd.DataFrame(
    {
        "name": ["apple", "banana", "apple", "orange"],
        "store": ["Aldi", "Walmart", "Walmart", np.nan],
        "price": [2, 1, 3, 4],
    }
)

schema = pa.DataFrameSchema(
    {
        "name": Column(str, Check.isin(available_fruits)),
        "store": Column(str, Check.isin(nearby_stores), nullable=True),
        "price": Column(int, Check.less_than(5)),
    }
)
schema.validate(fruits)
```

null.py hosted with ❤ by GitHub view raw

Deal with Duplicates

By default, duplicates are acceptable. To raise an error when there are duplicates, use `allow_duplicates=False` :

```python
schema = pa.DataFrameSchema(
    {
        "name": Column(str, Check.isin(available_fruits)),
        "store": Column(
            str, Check.isin(nearby_stores), nullable=True, allow_duplicates=False
        ),
        "price": Column(int, Check.less_than(5)),
    }
)
schema.validate(fruits)
```

duplicates.py hosted with ❤ by GitHub view raw

```
SchemaError: series 'store' contains duplicate values: {2: 'Walmart'}
```

Convert Data Types

`coerce=True` changes the data type of a column. If coercion is not possible, Pandera raises an error.

In the code below, the data type of price is changed from integer to string.

```python
fruits = pd.DataFrame(
    {
        "name": ["apple", "banana", "apple", "orange"],
        "store": ["Aldi", "Walmart", "Walmart", "Aldi"],
        "price": [2, 1, 3, 4],
    }
)

schema = pa.DataFrameSchema({"price": Column(str, coerce=True)})
validated = schema.validate(fruits)
validated.dtypes
```

convert_data_types.py hosted with ❤ by GitHub view raw

```
name     object
store    object
price    object
dtype: object
```

Match Patterns

What if we want to change all columns that start with the word `store` ?

```python
favorite_stores = ["Aldi", "Walmart", "Whole Foods", "Schnucks"]

fruits = pd.DataFrame(
    {
        "name": ["apple", "banana", "apple", "orange"],
        "store_nearby": ["Aldi", "Walmart", "Walmart", "Aldi"],
        "store_far": ["Whole Foods", "Schnucks", "Whole Foods", "Schnucks"],
    }
)
```

data.py hosted with ❤ by GitHub view raw

Pandera allows us to apply the same checks on multiple columns that share a certain pattern by adding `regex=True` :

```python
schema = pa.DataFrameSchema(
    {
        "name": Column(str, Check.isin(available_fruits)),
        "store_+": Column(str, Check.isin(favorite_stores), regex=True),
    }
)
schema.validate(fruits)
```

pattern_matching.py hosted with ❤ by GitHub view raw

Export and Load From a YAML File

Export to YAML

Using a YAML file is a neat way to show your tests to colleagues who don't know Python. We can keep a record of all validations in a YAML file using `schema.to_yaml()` :

The `schema.yml` should look like the below:

Load from YAML

To load from a YAML file, simple use `pa.io.from_yaml(yaml_schema)` :

Conclusion

Congratulations! You have just learned how to use Pandera to validate your dataset. Since data is an important aspect of a data science project, validating the inputs and outputs of your functions will reduce the errors down the pipeline.

Feel free to play and fork the source code of this article here:

Data-science/pandera.ipynb at master · khuyentran1401/Data-science

Collection of useful data science topics along with code and articles - Data-science/pandera.ipynb at master ·...

github.com

I like to write about basic data science concepts and play with different data science tools. You could connect with me on LinkedIn and Twitter.

Star this repo if you want to check out the codes for all of the articles I have written. Follow me on Medium to stay informed with my latest data science articles like these:

BentoML: Create an ML Powered Prediction Service in Minutes
Containerize and Deploy Your ML Model in Python
towardsdatascience.com

4 pre-commit Plugins to Automate Code Reviewing and Formatting in Python
Write High-Quality Code with black, flake8, isort, and interrogate
towardsdatascience.com

Orchestrate a Data Science Project in Python With Prefect
Optimize Your Data Science Workflow in a Few Lines of Code
towardsdatascience.com

Introduction to DVC: Data Version Control Tool for Machine Learning Projects
Just like Git, but with Data!
towardsdatascience.com

Data Data Science Test Python Editors Pick