

Pandera: Beyond Pandas Data Validation 🐼✅

Niels Bantilan, Chief ML Engineer @ Union.ai

SciPy 2023, July 12th 2023

Background

- 📖 B.A. in Biology and Dance
- 📖 M.P.H. in Sociomedical Science and Public Health Informatics
- 🤖 Chief Machine Learning Engineer @ Union.ai
- 🛠️ Flytekit OSS Maintainer
- ✅ Author and Maintainer of Pandera
- 💪 Author of UnionML
- 🔧 Make DS/ML practitioners more productive

This is a talk about open source development 🧑💻

Outline



- 🐣 Origins: solving a local problem
- 🐔 Evolution: solving other people's problems
- 🦩 Revolution: rewriting Pandera's internals
- 🌅 What's next?

Where's the Code?



Slides: https://unionai-oss.github.io/pandera-presentations/slides/20230712_scipy_beyond_pandas.slides.html



Notebook: https://github.com/unionai-oss/pandera-presentations/blob/master/notebooks/20230712_scipy_beyond_pandas.ipynb



Origins

 Why Should I Validate Data?

What's a DataFrame?

```
dataframe.head()
```

	hours_worked	wage_per_hour
person_id		
1014c40	38.5	15.1
9bdac94	41.25	15.0
4331eb8	35.0	21.3
ee7af68	27.75	17.5
d188ff6	22.25	19.5

What's Data Validation?

Data validation is the act of *falsifying* data against explicit assumptions for some downstream purpose, like analysis, modeling, and visualization.

What's Data Validation?

Data validation is the act of *falsifying* data against explicit assumptions for some downstream purpose, like analysis, modeling, and visualization.

"All swans are white"

What's Data Validation?

Data validation is the act of *falsifying* data against explicit assumptions for some downstream purpose, like analysis, modeling, and visualization.


"All swans are white"




CC BY-SA 3.0, [Link](#)


Why Do I Need it?

Why Do I Need it?

 It can be difficult to reason about and debug data processing pipelines.

Why Do I Need it?

 It can be difficult to reason about and debug data processing pipelines.

 It's critical to ensuring data quality in many contexts especially when the end product informs business decisions, supports scientific findings, or generates predictions in a production setting.

Everyone has a personal relationship with their data

Everyone has a personal relationship with their data

Story Time

Imagine that you're a data scientist maintaining an existing data processing pipeline   ...

One day, you encounter an error log trail and decide to follow it...

```
/usr/local/miniconda3/envs/pandera-presentations/lib/python3.7/site-packages/pandas/core/ops/__init__.py in  
masked_arith_op(x, y, op)  
    445     if mask.any():  
    446         with np.errstate(all="ignore"):  
--> 447             result[mask] = op(xrav[mask], com.values_from_object(yrav[mask]))  
    448  
    449     else:  
  
TypeError: can't multiply sequence by non-int of type 'float'
```

And you find yourself at the top of a function...

```
def process_data(df):  
    ...
```

You look around, and see some hints of what had happened...

```
def process_data(df):  
    return df.assign(weekly_income=lambda x: x.hours_worked * x.wage_per_hour)
```

You sort of know what's going on, but you want to take a closer look!

```
def process_data(df):  
    import pdb; pdb.set_trace()  # <- insert breakpoint  
    return df.assign(weekly_income=lambda x: x.hours_worked * x.wage_per_hour)
```

And you find some funny business going on...

```
print(df)
```

person_id	hours_worked	wage_per_hour
1014c40	38.5	15.1
9bdac94	41.25	15.0
4331eb8	35.0	21.3
ee7af68	27.75	17.5
d188ff6	22.25	19.5
a0c4b6e	-20.5	25.5

```
df.dtypes
```

```
hours_worked    object
wage_per_hour   float64
dtype: object
```

```
df.hours_worked.map(type)
```

```
person_id
1014c40    <class 'float'>
9bdac94    <class 'float'>
4331eb8    <class 'str'>
ee7af68    <class 'float'>
d188ff6    <class 'float'>
a0c4b6e    <class 'float'>
Name: hours_worked, dtype: object
```

You squash the bug and add documentation for the next weary traveler who happens upon this code.

```
def process_data(df):  
    return (  
        df  
        # make sure columns are floats  
        .astype({"hours_worked": float, "wage_per_hour": float})  
        # replace negative values with nans  
        .assign(hours_worked=lambda x: x.hours_worked.where(x.hours_worked >= 0, np.nan))  
        # compute weekly income  
        .assign(weekly_income=lambda x: x.hours_worked * x.wage_per_hour)  
    )
```

```
process_data(df)
```

	hours_worked	wage_per_hour	weekly_income
person_id			
1014c40	38.50	15.1	581.350
9bdac94	41.25	15.0	618.750
4331eb8	35.00	21.3	745.500
ee7af68	27.75	17.5	485.625
d188ff6	22.25	19.5	433.875
a0c4b6e	NaN	25.5	NaN

🕒 A few months later...

You find yourself at a familiar function, but it looks a little different from when you left it...

```
@pa.check_types
def process_data(df: DataFrame[RawData]) -> DataFrame[ProcessedData]:
    return (
        # replace negative values with nans
        df.assign(hours_worked=lambda x: x.hours_worked.where(x.hours_worked >= 0, np.nan))
        # compute weekly income
        .assign(weekly_income=lambda x: x.hours_worked * x.wage_per_hour)
    )
```


You look above and see what `RawData` and `ProcessedData` are, finding a `NOTE` that a fellow traveler has left for you.

```
import pandera as pa

# NOTE: this is what's supposed to be in `df` going into `process_data`
class RawData(pa.SchemaModel):
    hours_worked: float = pa.Field(coerce=True, nullable=True)
    wage_per_hour: float = pa.Field(coerce=True, nullable=True)

# ... and this is what `process_data` is supposed to return.
class ProcessedData(RawData):
    hours_worked: float = pa.Field(coerce=True, nullable=True)
    weekly_income: float = pa.Field(nullable=True)

@pa.check_types
def process_data(df: DataFrame[RawData]) -> DataFrame[ProcessedData]:
    ...
```

Moral of the Story

Moral of the Story

The better you can reason about the contents of a dataframe, the faster you can debug.

Moral of the Story

The better you can reason about the contents of a dataframe, the faster you can debug.

The faster you can debug, the sooner you can focus on downstream tasks that you care about.

Moral of the Story

The better you can reason about the contents of a dataframe, the faster you can debug.

The faster you can debug, the sooner you can focus on downstream tasks that you care about.

By validating data through explicit contracts, you also create data documentation *and* a simple, stateless data shift detector.

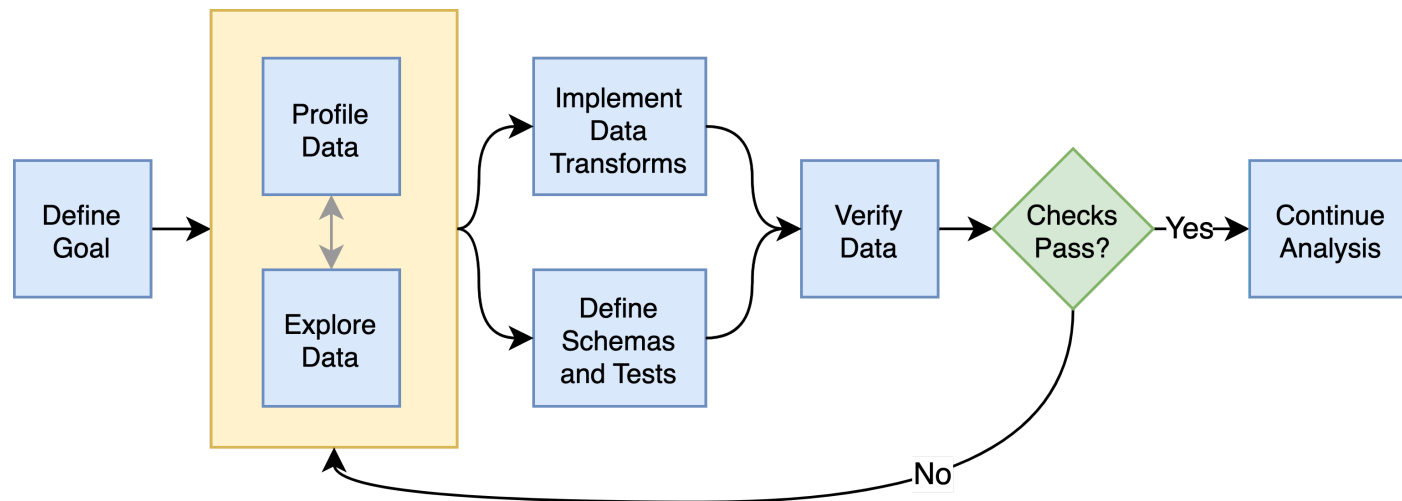
Pandera Design Principles

From [scipy 2020 - pandera: Statistical Data Validation of Pandas Dataframes](#)

- Expressing validation rules should feel familiar to pandas users.
- Data validation should be compatible with the different workflows and tools in the data science toolbelt without a lot of setup or configuration.
- Defining custom validation rules should be easy.
- The validation interface should make the debugging process easier.
- Integration with existing code should be as seamless as possible.

Pandera Programming Model

The pandera programming model is an iterative loop of building statistical domain knowledge, implementing data transforms and schemas, and verifying data.



Meta Comment

This presentation notebook is validated by pandera 🤖





What's Data Testing

And How Can I Put it Into Practice?

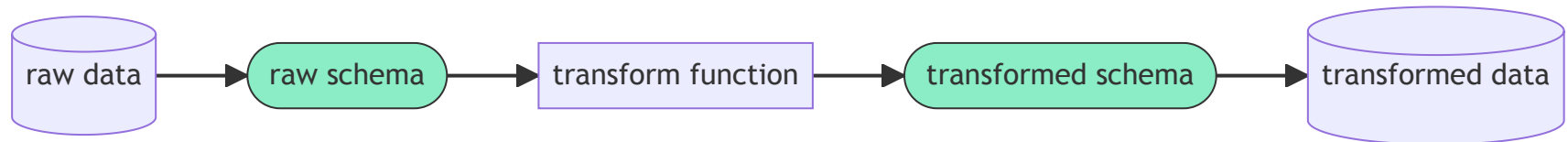
Data validation: *The act of falsifying data against explicit assumptions for some downstream purpose, like analysis, modeling, and visualization.*

Data validation: *The act of falsifying data against explicit assumptions for some downstream purpose, like analysis, modeling, and visualization.*

Data Testing: *Validating not only real data, but also the functions that produce them.*

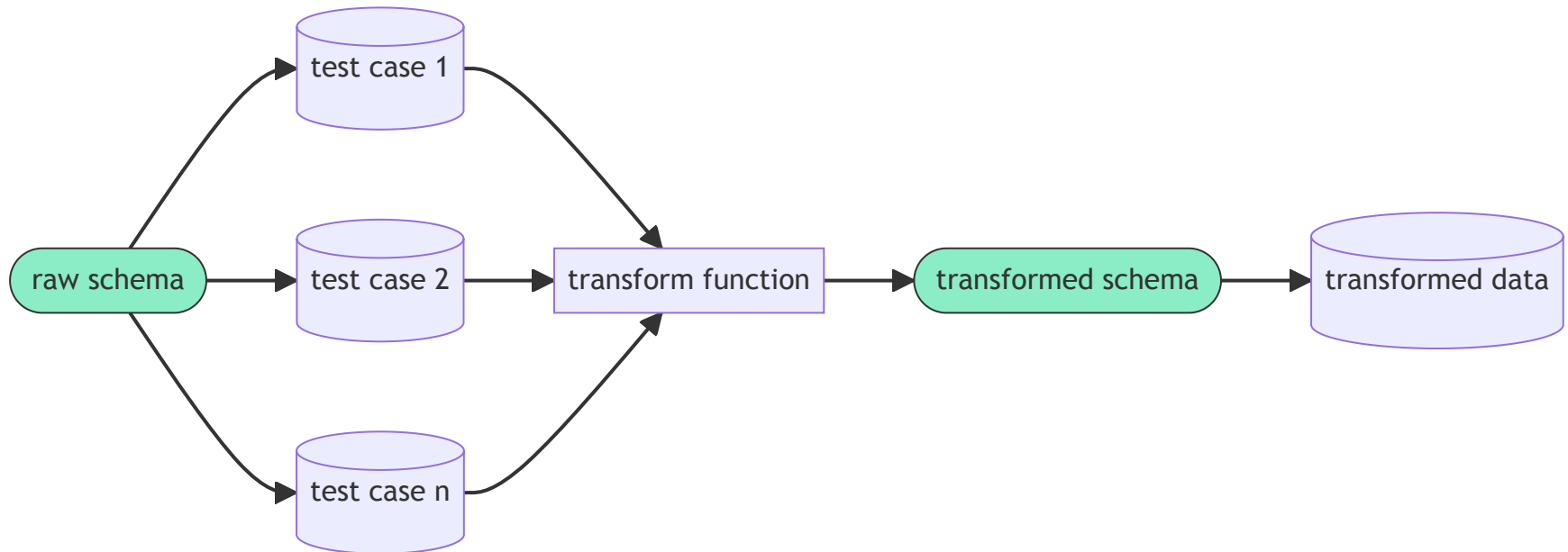
In the Real World

Validate real data in production



In the Test Suite

Validate functions that produce data, given some test cases





Pandera

An expressive and light-weight statistical typing tool for dataframe-like containers

- Check the types and properties of dataframes
- Easily integrate with existing data pipelines via function decorators
- Synthesize data from schema objects for property-based testing

Object-based API

Defining a schema looks and feels like defining a pandas dataframe

```
import pandera as pa

clean_data_schema = pa.DataFrameSchema(
    columns={
        "continuous": pa.Column(float, pa.Check.ge(0), nullable=True),
        "categorical": pa.Column(str, pa.Check.isin(["A", "B", "C"]), nullable=True),
    },
    coerce=True,
)
```

Object-based API

Defining a schema looks and feels like defining a pandas dataframe

```
import pandera as pa

clean_data_schema = pa.DataFrameSchema(
    columns={
        "continuous": pa.Column(float, pa.Check.ge(0), nullable=True),
        "categorical": pa.Column(str, pa.Check.isin(["A", "B", "C"]), nullable=True),
    },
    coerce=True,
)
```

Class-based API

Define complex types with modern Python, inspired by [pydantic](#) and `dataclasses`

```
from pandera.typing import DataFrame, Series

class CleanData(pa.SchemaModel):
    continuous: Series[float] = pa.Field(ge=0, nullable=True)
    categorical: Series[str] = pa.Field(isin=["A", "B", "C"], nullable=True)

    class Config:
        coerce = True
```


Pandera Raises Informative Errors

Know Exactly What Went Wrong with Your Data

```
raw_data = pd.DataFrame({
    "continuous": [-1.1, "4.0", "10.25", "-0.1", "5.2"],
    "categorical": ["A", "B", "C", "Z", "X"],
})

try:
    CleanData.validate(raw_data, lazy=True)
except pa.errors.SchemaErrors as exc:
    display(exc.failure_cases)
```

	schema_context	column	check	check_number	failure_case	index
0	Column	continuous	greater_than_or_equal_to(0)	0	-1.1	0
1	Column	continuous	greater_than_or_equal_to(0)	0	-0.1	3
2	Column	categorical	isin(['A', 'B', 'C'])	0	Z	3
3	Column	categorical	isin(['A', 'B', 'C'])	0	X	4

Pandera Supports Schema Transformations/Inheritance

Object-based API

Dynamically transform schema objects on the fly

```
raw_data_schema = pa.DataFrameSchema(  
    columns={  
        "continuous": pa.Column(float),  
        "categorical": pa.Column(str),  
    },  
    coerce=True,  
)  
  
clean_data_schema.update_columns({  
    "continuous": {"nullable": True},  
    "categorical": {"checks": pa.Check.isin(["A", "B", "C"]), "nullable": True},  
});
```

Pandera Supports Schema Transformations/Inheritance

Object-based API

Dynamically transform schema objects on the fly

```
raw_data_schema = pa.DataFrameSchema(  
    columns={  
        "continuous": pa.Column(float),  
        "categorical": pa.Column(str),  
    },  
    coerce=True,  
)  
  
clean_data_schema.update_columns({  
    "continuous": {"nullable": True},  
    "categorical": {"checks": pa.Check.isin(["A", "B", "C"]), "nullable": True},  
});
```

Class-based API

Inherit from `pandera.SchemaModel` to Define Type Hierarchies

```
class RawData(pa.SchemaModel):  
    continuous: Series[float]  
    categorical: Series[str]  
  
    class Config:  
        coerce = True  
  
class CleanData(RawData):  
    continuous = pa.Field(ge=0, nullable=True)  
    categorical = pa.Field(isin=["A", "B", "C"], nullable=True);
```

Integrate Seamlessly with your Pipeline

Use decorators to add IO checkpoints to the critical functions in your pipeline

```
@pa.check_types
def fn(raw_data: DataFrame[RawData]) -> DataFrame[CleanData]:
    return raw_data.assign(
        continuous=lambda df: df["continuous"].where(lambda x: x > 0, np.nan),
        categorical=lambda df: df["categorical"].where(lambda x: x.isin(["A", "B", "C"]), np.nan),
    )

fn(raw_data)
```

	continuous	categorical
0	NaN	A
1	4.00	B
2	10.25	C
3	NaN	NaN
4	5.20	NaN

Generative Schemas

Schemas that synthesize valid data under its constraints

```
CleanData.example(size=5)
```

	continuous	categorical
0	NaN	A
1	NaN	A
2	NaN	C
3	4.501643e+15	NaN
4	NaN	C

Generative Schemas

Schemas that synthesize valid data under its constraints

```
CleanData.example(size=5)
```

	continuous	categorical
0	NaN	A
1	NaN	A
2	NaN	C
3	4.501643e+15	NaN
4	NaN	C

Data Testing: Test the functions that produce clean data

```
from hypothesis import given

@given(RawData.strategy(size=5))
def test_fn(raw_data):
    fn(raw_data)

def run_test_suite():
    test_fn()
    print("tests passed ✅")

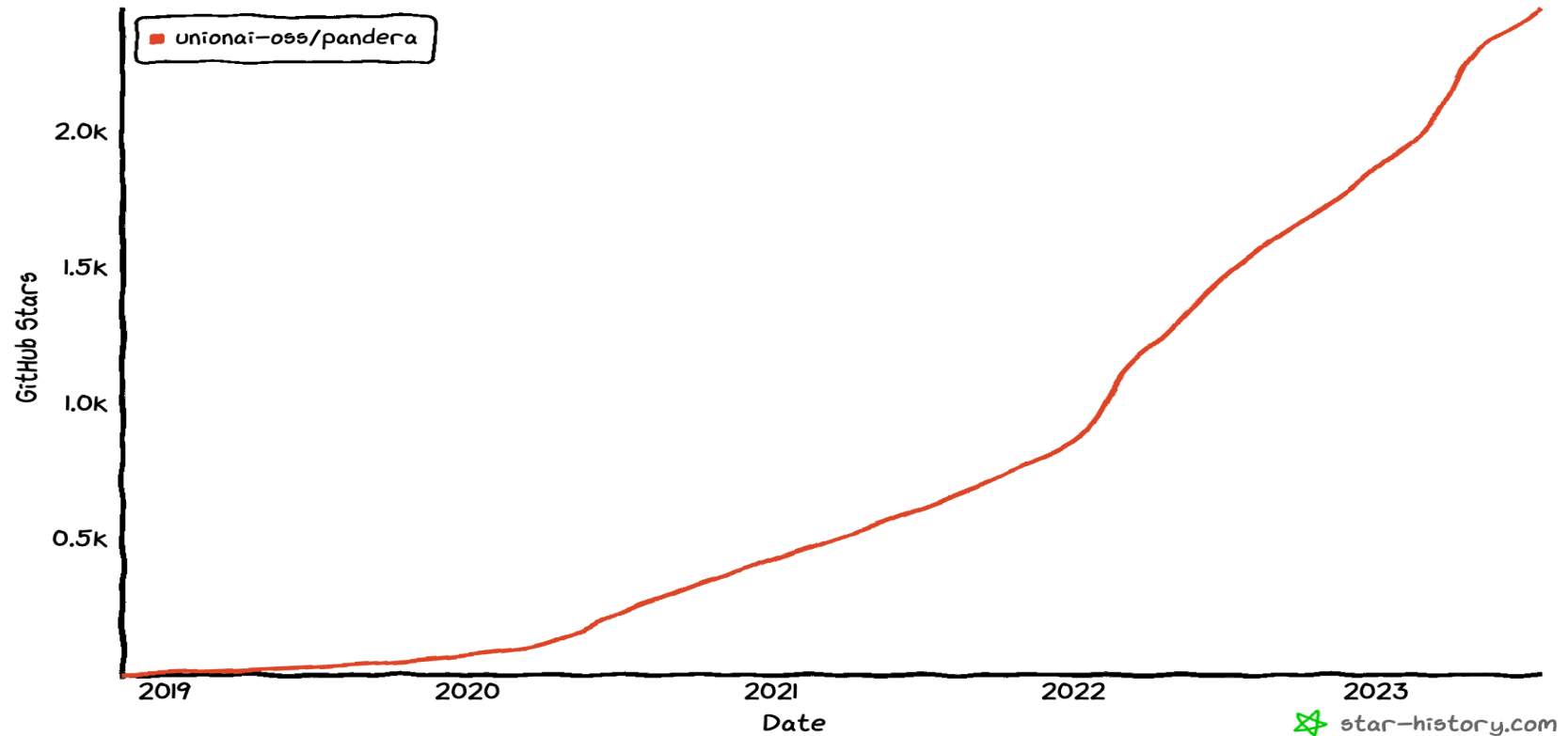
run_test_suite()
```

tests passed ✅



Evolution

🌟 Star History



Major Events

Major Events

 Documentation Improvements


Major Events

 Documentation Improvements

 Class-based API

Major Events


 Documentation Improvements

 Class-based API

 Data Synthesis Strategies

Major Events

 Documentation Improvements

 Class-based API

 Data Synthesis Strategies

 Pandera Type System

Expanding scope

Adding `geopandas`, `dask`, `modin`, and `pyspark.pandas` was relatively straight forward.

```
display(raw_data)
```

	continuous	categorical
0	-1.1	A
1	4.0	B
2	10.25	C
3	-0.1	Z
4	5.2	X

dask

```
import dask.dataframe as dd

dask_dataframe = dd.from_pandas(raw_data, npartitions=1)

try:
    CleanData(dask_dataframe, lazy=True).compute()
except pa.errors.SchemaErrors as exc:
    display(exc.failure_cases.sort_index())
```

	schema_context	column	check	check_number	failure_case	index
0	Column	continuous	greater_than_or_equal_to(0)	0	-1.1	0
1	Column	continuous	greater_than_or_equal_to(0)	0	-0.1	3
2	Column	categorical	isin(['A', 'B', 'C'])	0	Z	3
3	Column	categorical	isin(['A', 'B', 'C'])	0	X	4

modin

```
import modin.pandas as mpd

modin_dataframe = mpd.DataFrame(raw_data)

try:
    CleanData(modin_dataframe, lazy=True)
except pa.errors.SchemaErrors as exc:
    display(exc.failure_cases.sort_index())
```

	schema_context	column	check	check_number	failure_case	index
0	Column	continuous	greater_than_or_equal_to(0)	0	-1.1	0
1	Column	continuous	greater_than_or_equal_to(0)	0	-0.1	3
2	Column	categorical	isin(['A', 'B', 'C'])	0	Z	3
3	Column	categorical	isin(['A', 'B', 'C'])	0	X	4

pyspark.pandas

```
import pyspark.pandas as ps

pyspark_pd_dataframe = ps.DataFrame(raw_data)

try:
    CleanData(pyspark_pd_dataframe, lazy=True)
except pa.errors.SchemaErrors as exc:
    display(exc.failure_cases)
```

	schema_context	column	check	check_number	failure_case	index
0	Column	continuous	greater_than_or_equal_to(0)	0	-1.1	0
1	Column	continuous	greater_than_or_equal_to(0)	0	-0.1	3
2	Column	categorical	isin(['A', 'B', 'C'])	0	Z	3
3	Column	categorical	isin(['A', 'B', 'C'])	0	X	4

Problem: What about non-pandas-compliant dataframes?

Problem: What about non-pandas-compliant dataframes?

Design weaknesses

- Schemas and checks were strongly coupled with pandas
- Error reporting and eager validation assumed in-memory data
- Leaky pandas abstractions

Problem: What about non-pandas-compliant dataframes?

Design weaknesses

- Schemas and checks were strongly coupled with pandas
- Error reporting and eager validation assumed in-memory data
- Leaky pandas abstractions

Design strengths

- Generic schema interface
- Flexible check abstraction
- Flexible type system



Revolution

Re-writing pandera internals

High-level approach: decoupling schema specification from backend

- A `pandera.api` subpackage, which contains the schema specification that defines the properties of an underlying data structure.
- A `pandera.backends` subpackage, which leverages the schema specification and implements the actual validation logic.
- A backend registry, which maps a particular API specification to a backend based on the DataFrame type being validated.
- A common type-aware `Check` namespace and registry, which registers type-specific implementations of built-in checks and allows contributors to easily add new built-in checks.


Writing your own schema

```
import sloth as sl
from pandera.api.base.schema import BaseSchema
from pandera.backends.base import BaseSchemaBackend

class DataFrameSchema(BaseSchema):
    def __init__(self, **kwargs):
        # add properties that this dataframe would contain

class DataFrameSchemaBackend(BaseSchemaBackend):
    def validate(
        self,
        check_obj: sl.DataFrame,
        schema: DataFrameSchema,
        *,
        **kwargs,
    ):
        # implement custom validation logic

# register the backend
DataFrameSchema.register_backend(
    sloth.DataFrame,
    DataFrameSchemaBackend,
)
```

 Pandera now supports `pyspark.sql.DataFrame` in `0.16.0b!`

<https://pandera.readthedocs.io/en/latest/>

```
import pandera.pyspark as pa
import pyspark.sql.types as T

from decimal import Decimal
from pyspark.sql import DataFrame
from pandera.pyspark import DataFrameModel

class PanderaSchema(DataFrameModel):
    id: T.IntegerType() = pa.Field(gt=5)
    product_name: T.StringType() = pa.Field(str_startswith="B")
    price: T.DecimalType(20, 5) = pa.Field()
    description: T.ArrayType(T.StringType()) = pa.Field()
    meta: T.MapType(T.StringType(), T.StringType()) = pa.Field()
```

Organizational and Development Challenges

- Multi-tasking the rewrite with PR reviews
- Centralized knowledge
- Informal governance

Retrospective

Things in place that reduced the risk of regressions

- *Unit tests.*
- *Localized pandas coupling.*
- *Lessons learned from pandas-compliant integrations.*

Retrospective

Things in place that reduced the risk of regressions

- *Unit tests.*
- *Localized pandas coupling.*
- *Lessons learned from pandas-compliant integrations.*

Additional approaches to put into practice in the future:

- *Thoughtful design work.*
- *Library-independent error reporting.*
- *Decoupling metadata from data.*
- *Investing in governance and community.*

Updated Principles

```
diff --git a/pandera-principles.md b/pandera-principles.md
index 3e32552..7cf3392 100644
--- a/pandera-principles.md
+++ b/pandera-principles.md
@@ -1,7 +1,10 @@
-- Expressing schemas as code should feel familiar to pandas users.
+- Expressing schemas as code should feel familiar to Python users, regardless of
+ the dataframe library they're using.
- Data validation should be compatible with the different
  workflows and tools in the data science and ML stack
  without a lot of setup or configuration.
- Defining custom validation rules should be easy.
- The validation interface should make the debugging process easier.
- Integration with existing code should be as seamless as possible.
+- Extending the interface to other statistical data structures should be easy
+ using a core set of building blocks and abstractions.
(END)
```

Announcement

 Pandera has joined Union.ai 

Announcement

 Pandera has joined Union.ai 

What does this mean?

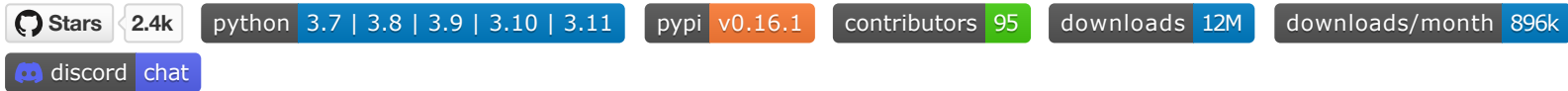
- It will continue to be open source.
- It will have more resources to maintain and govern it.
- We can learn from enterprise users.



Roadmap

- 🤝 **Integrations:** support more data manipulation libraries:
 - Polars support: <https://github.com/unionai-oss/pandera/issues/1064>
 - Ibis support: <https://github.com/unionai-oss/pandera/issues/1105>
 - Investigate the dataframe-api standard: <https://github.com/data-apis/dataframe-api>
 - Open an issue! <https://github.com/unionai-oss/pandera/issues>
- 🖥️ **User Experience:** polish the API:
 - better error-reporting
 - more built-in checks
 - conditional validation
- 🤝 **Interoperability:** tighter integrations with the python ecosystem:
 - `pydantic v2`
 - `pytest` : collect data coverage statistics
 - `hypothesis` : faster data synthesis
- 🏆 **Innovations:** new capabilities:
 - stateful data validation
 - model-based types

Join the Community!



- **Twitter:** [@cosmicbboy](#)
- **Discord:** <https://discord.gg/vyanhWuaKB>
- **Email:** niels@union.ai
- **Repo:** <https://github.com/unionai-oss/pandera>
- **Docs:** <https://pandera.readthedocs.io>
- **Contributing Guide:** <https://pandera.readthedocs.io/en/stable/CONTRIBUTING.html>
- **Become a Sponsor:** <https://github.com/sponsors/cosmicBboy>

Join me at the sprints!

Contribute to the `Flyte` project: <https://www.flyte.org>

```
import pandas as pd
from flytekit import Resources, task, workflow
from sklearn.datasets import load_wine
from sklearn.linear_model import LogisticRegression

@task(requests=Resources(mem="700Mi"))
def get_data() -> pd.DataFrame:
    """Get the wine dataset."""
    return load_wine(as_frame=True).frame

@task
def process_data(data: pd.DataFrame) -> pd.DataFrame:
    """Simplify the task from a 3-class to a binary classification problem."""
    return data.assign(target=lambda x: x["target"].where(x["target"] == 0, 1))

@task
def train_model(data: pd.DataFrame, hyperparameters: dict) -> LogisticRegression:
    """Train a model on the wine dataset."""
    features = data.drop("target", axis="columns")
```

Copy

