pandera

Niels Bantilan, Nigel Markey

Sep 21, 2020

TABLE OF CONTENTS

A data validation library for scientists, engineers, and analysts seeking correctness.

`pandera` provides a flexible and expressive API for performing data validation on tidy (long-form) and wide data to make data processing pipelines more readable and robust.

pandas data structures contain information that `pandera` explicitly validates at runtime. This is useful in production-critical data pipelines or reproducible research settings. With `pandera`, you can:

1. *Check* the types and properties of columns in a `pd.DataFrame` or values in a `pd.Series`.
2. Perform more complex statistical validation like *hypothesis testing*.
3. Seamlessly integrate with existing data analysis/processing pipelines via *function decorators*.

CHAPTER

ONE

INSTALL

Install with *pip*:

```
pip install pandera
```

Or conda:

```
conda install -c conda-forge pandera
```

CHAPTER

TWO

QUICK START

```
import pandas as pd
import pandera as pa

# data to validate
df = pd.DataFrame({
    "column1": [1, 4, 0, 10, 9],
    "column2": [-1.3, -1.4, -2.9, -10.1, -20.4],
    "column3": ["value_1", "value_2", "value_3", "value_2", "value_1"],
})

# define schema
schema = pa.DataFrameSchema({
    "column1": pa.Column(int, checks=pa.Check.less_than_or_equal_to(10)),
    "column2": pa.Column(float, checks=pa.Check.less_than(-1.2)),
    "column3": pa.Column(str, checks=[
        pa.Check.str_startswith("value_"),
        # define custom checks as functions that take a series as input and
        # outputs a boolean or boolean Series
        pa.Check(lambda s: s.str.split("_", expand=True).shape[1] == 2)
    ]),
})

validated_df = schema(df)
print(validated_df)
```

```
   column1  column2  column3
0        1     -1.3  value_1
1        4     -1.4  value_2
2        0     -2.9  value_3
3       10    -10.1  value_2
4        9    -20.4  value_1
```

Alternatively, you can pass the built-in python types that are supported by pandas, or strings representing the legal pandas datatypes:

```
schema = pa.DataFrameSchema({
    # built-in python types
    "int_column": pa.Column(int),
    "float_column": pa.Column(float),
    "str_column": pa.Column(str),

    # pandas dtype string aliases
    "int_column2": pa.Column("int64"),
```

(continues on next page)

(continued from previous page)

```
    "float_column2": pa.Column("float64"),
    # pandas > 1.0.0 support native "string" type
    "str_column2": pa.Column("object"),
})
```

For more details on data types, see *pandera.PandasDtype*

CHAPTER

THREE

INFORMATIVE ERRORS

If the dataframe does not pass validation checks, `pandera` provides useful error messages. An `error` argument can also be supplied to `Check` for custom error messages.

In the case that a validation `Check` is violated:

```
import pandas as pd

from pandera import Column, DataFrameSchema, Int, Check

simple_schema = DataFrameSchema({
    "column1": Column(
        Int, Check(lambda x: 0 <= x <= 10, element_wise=True,
                   error="range checker [0, 10]"))
})

# validation rule violated
fail_check_df = pd.DataFrame({
    "column1": [-20, 5, 10, 30],
})

simple_schema(fail_check_df)
```

```
Traceback (most recent call last):
...
SchemaError: <Schema Column: 'column1' type=int> failed element-wise validator 0:
<Check <lambda>: range checker [0, 10]>
failure cases:
   index  failure_case
0      0           -20
1      3            30
```

And in the case of a mis-specified column name:

```
# column name mis-specified
wrong_column_df = pd.DataFrame({
   "foo": ["bar"] * 10,
   "baz": [1] * 10
})

simple_schema.validate(wrong_column_df)
```

```
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
pandera.SchemaError: column 'column1' not in dataframe
   foo  baz
0  bar    1
1  bar    1
2  bar    1
3  bar    1
4  bar    1
```

CHAPTER

FOUR

CONTRIBUTING

All contributions, bug reports, bug fixes, documentation improvements, enhancements and ideas are welcome.

A detailed overview on how to contribute can be found in the contributing guide on GitHub.

CHAPTER

FIVE

ISSUES

Submit issues, feature requests or bugfixes on github.

5.1 DataFrame Schemas

The *DataFrameSchema* class enables the specification of a schema that verifies the columns and index of a pandas `DataFrame` object.

The `DataFrameSchema` object consists of *Column*s and an *Index*.

```
import pandera as pa

from pandera import Column, DataFrameSchema, Check, Index

schema = DataFrameSchema(
    {
        "column1": Column(pa.Int),
        "column2": Column(pa.Float, Check(lambda s: s < -1.2)),
        # you can provide a list of validators
        "column3": Column(pa.String, [
           Check(lambda s: s.str.startswith("value")),
           Check(lambda s: s.str.split("_", expand=True).shape[1] == 2)
        ]),
    },
    index=Index(pa.Int),
    strict=True,
    coerce=True,
)
```

5.1.1 Column Validation

A *Column* must specify the properties of a column in a dataframe object. It can be optionally verified for its data type, *null values* or duplicate values. The column can be *coerced* into the specified type, and the *required* parameter allows control over whether or not the column is allowed to be missing.

Column checks allow for the DataFrame's values to be checked against a user-provided function. `Check` objects also support *grouping* by a different column so that the user can make assertions about subsets of the column of interest.

Column Hypotheses enable you to perform statistical hypothesis tests on a DataFrame in either wide or tidy format. See *Hypothesis Testing* for more details.

5.1.1.1 Null Values in Columns

By default, SeriesSchema/Column objects assume that values are not nullable. In order to accept null values, you need to explicitly specify `nullable=True`, or else you'll get an error.

```
import numpy as np
import pandas as pd
import pandera as pa

from pandera import Check, Column, DataFrameSchema

df = pd.DataFrame({"column1": [5, 1, np.nan]})

non_null_schema = DataFrameSchema({
    "column1": Column(pa.Int, Check(lambda x: x > 0))
})

non_null_schema.validate(df)
```

```
Traceback (most recent call last):
...
SchemaError: non-nullable series contains null values: {2: nan}
```

Note: Due to a known limitation in pandas prior to version 0.24.0, integer arrays cannot contain `NaN` values, so this schema will return a DataFrame where `column1` is of type `float`. *`PandasDtype`* does not currently support the nullable integer array type, but you can still use the "Int64" string alias for nullable integer arrays

```
null_schema = DataFrameSchema({
    "column1": Column(pa.Int, Check(lambda x: x > 0), nullable=True)
})

print(null_schema.validate(df))
```

```
   column1
0      5.0
1      1.0
2      NaN
```

5.1.1.2 Coercing Types on Columns

If you specify `Column(dtype, ..., coerce=True)` as part of the DataFrameSchema definition, calling `schema.validate` will first coerce the column into the specified `dtype` before applying validation checks.

```
import pandas as pd
import pandera as pa

from pandera import Column, DataFrameSchema

df = pd.DataFrame({"column1": [1, 2, 3]})
schema = DataFrameSchema({"column1": Column(pa.String, coerce=True)})

validated_df = schema.validate(df)
assert isinstance(validated_df.column1.iloc[0], str)
```

Note: Note the special case of integers columns not supporting `nan` values. In this case, `schema.validate` will complain if `coerce == True` and null values are allowed in the column.

```
df = pd.DataFrame({"column1": [1., 2., 3, np.nan]})
schema = DataFrameSchema({
    "column1": Column(pa.Int, coerce=True, nullable=True)
})

validated_df = schema.validate(df)
```

```
Traceback (most recent call last):
...
ValueError: cannot convert float NaN to integer
```

The best way to handle this case is to simply specify the column as a `Float` or `Object`.

```
schema_object = DataFrameSchema({
    "column1": Column(pa.Object, coerce=True, nullable=True)
})
schema_float = DataFrameSchema({
    "column1": Column(pa.Float, coerce=True, nullable=True)
})

print(schema_object.validate(df).dtypes)
print(schema_float.validate(df).dtypes)
```

```
column1    object
dtype: object
column1    float64
dtype: object
```

If you want to coerce all of the columns specified in the `DataFrameSchema`, you can specify the `coerce` argument with `DataFrameSchema(..., coerce=True)`.

5.1.1.3 Required Columns

By default all columns specified in the schema are required, meaning that if a column is missing in the input DataFrame an exception will be thrown. If you want to make a column optional, specify `required=False` in the column constructor:

```
import pandas as pd
import pandera as pa

from pandera import Column, DataFrameSchema

df = pd.DataFrame({"column2": ["hello", "pandera"]})
schema = DataFrameSchema({
    "column1": Column(pa.Int, required=False),
    "column2": Column(pa.String)
})

validated_df = schema.validate(df)
print(validated_df)
```

```
   column2
0    hello
1  pandera
```

Since `required=True` by default, missing columns would raise an error:

```
schema = DataFrameSchema({
    "column1": Column(pa.Int),
    "column2": Column(pa.String),
})

schema.validate(df)
```

```
Traceback (most recent call last):
...
pandera.SchemaError: column 'column1' not in dataframe
   column2
0    hello
1  pandera
```

5.1.1.4 Stand-alone Column Validation

In addition to being used in the context of a `DataFrameSchema`, `Column` objects can also be used to validate columns in a dataframe on its own:

```
import pandas as pd
import pandera as pa

df = pd.DataFrame({
    "column1": [1, 2, 3],
    "column2": ["a", "b", "c"],
})

column1_schema = pa.Column(pa.Int, name="column1")
column2_schema = pa.Column(pa.String, name="column2")

# pass the dataframe as an argument to the Column object callable
df = column1_schema(df)
validated_df = column2_schema(df)

# or explicitly use the validate method
df = column1_schema.validate(df)
validated_df = column2_schema.validate(df)

# use the DataFrame.pipe method to validate two columns
validated_df = df.pipe(column1_schema).pipe(column2_schema)
```

For multi-column use cases, the `DataFrameSchema` is still recommended, but if you have one or a small number of columns to verify, using `Column` objects by themselves is appropriate.

5.1.1.5 Column Regex Pattern Matching

In the case that your dataframe has multiple columns that share common statistical properties, you might want to specify a regex pattern that matches a set of meaningfully grouped columns that have `str` names.

```
import numpy as np
import pandas as pd
import pandera as pa

categories = ["A", "B", "C"]

np.random.seed(100)

dataframe = pd.DataFrame({
    "cat_var_1": np.random.choice(categories, size=100),
    "cat_var_2": np.random.choice(categories, size=100),
    "num_var_1": np.random.uniform(0, 10, size=100),
    "num_var_2": np.random.uniform(20, 30, size=100),
})

schema = pa.DataFrameSchema({
    "num_var_*": pa.Column(
        pa.Float,
        checks=pa.Check.greater_than_or_equal_to(0),
        regex=True,
    ),
    "cat_var_*": pa.Column(
        pa.Category,
        checks=pa.Check.isin(categories),
        coerce=True,
        regex=True,
    ),
})

print(schema.validate(dataframe).head())
```

```
  cat_var_1 cat_var_2  num_var_1  num_var_2
0         A         A   6.804147  24.743304
1         A         C   3.684308  22.774633
2         A         C   5.911288  28.416588
3         C         A   4.790627  21.951250
4         C         B   4.504166  28.563142
```

You can also regex pattern match on `pd.MultiIndex` columns:

```
np.random.seed(100)

dataframe = pd.DataFrame({
    ("cat_var_1", "y1"): np.random.choice(categories, size=100),
    ("cat_var_2", "y2"): np.random.choice(categories, size=100),
    ("num_var_1", "x1"): np.random.uniform(0, 10, size=100),
    ("num_var_2", "x2"): np.random.uniform(0, 10, size=100),
})

schema = pa.DataFrameSchema({
    ("num_var_*", "x*"): pa.Column(
        pa.Float,
```

(continues on next page)

(continued from previous page)

```
        checks=pa.Check.greater_than_or_equal_to(0),
        regex=True,
    ),
    ("cat_var_*", "y*"): pa.Column(
        pa.Category,
        checks=pa.Check.isin(categories),
        coerce=True,
        regex=True,
    ),
})

print(schema.validate(dataframe).head())
```

```
  cat_var_1 cat_var_2 num_var_1 num_var_2
         y1        y2        x1        x2
0          A         A  6.804147  4.743304
1          A         C  3.684308  2.774633
2          A         C  5.911288  8.416588
3          C         A  4.790627  1.951250
4          C         B  4.504166  8.563142
```

5.1.1.6 Handling Dataframe Columns not in the Schema

By default, columns that aren't specified in the schema aren't checked. If you want to check that the DataFrame *only* contains columns in the schema, specify `strict=True`:

```
import pandas as pd
import pandera as pa

from pandera import Column, DataFrameSchema

schema = DataFrameSchema(
    {"column1": Column(pa.Int)},
    strict=True)

df = pd.DataFrame({"column2": [1, 2, 3]})

schema.validate(df)
```

```
Traceback (most recent call last):
...
SchemaError: column 'column2' not in DataFrameSchema {'column1': <Schema Column: 'None
↪' type=int>}
```

5.1.2 Index Validation

You can also specify an *Index* in the *DataFrameSchema*.

```
import pandas as pd
import pandera as pa

from pandera import Column, DataFrameSchema, Index, Check

schema = DataFrameSchema(
   columns={"a": Column(pa.Int)},
   index=Index(
       pa.String,
       Check(lambda x: x.str.startswith("index_"))))

df = pd.DataFrame(
    data={"a": [1, 2, 3]},
    index=["index_1", "index_2", "index_3"])

print(schema.validate(df))
```

```
         a
index_1  1
index_2  2
index_3  3
```

In the case that the DataFrame index doesn't pass the `Check`.

```
df = pd.DataFrame(
    data={"a": [1, 2, 3]},
    index=["foo1", "foo2", "foo3"])

schema.validate(df)
```

```
Traceback (most recent call last):
...
SchemaError: <Schema Index> failed element-wise validator 0:
<lambda>
failure cases:
             index  count
failure_case
foo1           [0]      1
foo2           [1]      1
foo3           [2]      1
```

5.1.3 MultiIndex Validation

`pandera` also supports multi-index column and index validation.

5.1.3.1 MultiIndex Columns

Specifying multi-index columns follows the `pandas` syntax of specifying tuples for each level in the index hierarchy:

```
import pandas as pd
import pandera as pa

from pandera import Column, DataFrameSchema, Index

schema = DataFrameSchema({
    ("foo", "bar"): Column(pa.Int),
    ("foo", "baz"): Column(pa.String)
})

df = pd.DataFrame({
    ("foo", "bar"): [1, 2, 3],
    ("foo", "baz"): ["a", "b", "c"],
})

print(schema.validate(df))
```

```
  foo
  bar baz
0   1   a
1   2   b
2   3   c
```

5.1.3.2 MultiIndex Indexes

The *pandera.MultiIndex* class allows you to define multi-index indexes by composing a list of `pandera.Index` objects.

```
import pandas as pd
import pandera as pa

from pandera import Column, DataFrameSchema, Index, MultiIndex, Check

schema = DataFrameSchema(
    columns={"column1": Column(pa.Int)},
    index=MultiIndex([
        Index(pa.String,
              Check(lambda s: s.isin(["foo", "bar"])),
              name="index0"),
        Index(pa.Int, name="index1"),
    ])
)

df = pd.DataFrame(
    data={"column1": [1, 2, 3]},
    index=pd.MultiIndex.from_arrays(
        [["foo", "bar", "foo"], [0, 1,2 ]],
```

(continues on next page)

(continued from previous page)

```
        names=["index0", "index1"]
    )
)

print(schema.validate(df))
```

```
               column1
index0 index1
foo    0             1
bar    1             2
foo    2             3
```

5.1.4 Get Pandas Datatypes

Pandas provides a *dtype* parameter for casting a dataframe to a specific dtype schema. `DataFrameSchema` provides a *dtype* property which returns a pandas style dict. The keys of the dict are column names and values are the dtype.

Some examples of where this can be provided to pandas are:

- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html
- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.astype.html

```
import pandas as pd
import pandera as pa

schema = pa.DataFrameSchema(
    columns={
      "column1": pa.Column(pa.Int),
      "column2": pa.Column(pa.Category),
      "column3": pa.Column(pa.Bool)
    },
)

df = pd.DataFrame.from_dict(
  {
      "a": {"column1": 1, "column2": "valueA", "column3": True},
      "b": {"column1": 1, "column2": "valueB", "column3": True},
  },
  orient="index"
).astype(schema.dtype).sort_index(axis=1)

print(schema.validate(df))
```

```
   column1 column2  column3
a        1  valueA     True
b        1  valueB     True
```

5.1.5 DataFrameSchema Transformations

Pandera supports transforming a schema using *DataFrameSchema.add_columns()* and *DataFrameSchema.remove_columns()*.

Once you've defined a schema, you can add columns to the schema and to create a new schema object with the additional columns. This is useful for re-using schema objects in a data pipeline when additional computation has been done on a dataframe, therefore requiring additional checks.

```
import pandas as pd
import pandera as pa

data = pd.DataFrame({"col1": range(1, 6)})

schema = pa.DataFrameSchema(
    columns={"col1": pa.Column(pa.Int, pa.Check(lambda s: s >= 0))},
    strict=True)

transformed_schema = schema.add_columns({
    "col2": pa.Column(pa.String, pa.Check(lambda s: s == "value")),
    "col3": pa.Column(pa.Float, pa.Check(lambda x: x == 0.0)),
})

# validate original data
data = schema.validate(data)

# transformation
transformed_data = data.assign(col2="value", col3=0.0)

# validate transformed data
print(transformed_schema.validate(transformed_data))
```

```
   col1   col2  col3
0     1  value   0.0
1     2  value   0.0
2     3  value   0.0
3     4  value   0.0
4     5  value   0.0
```

Similarly, if you want dropped columns to be explicitly validated in a data pipeline:

```
import pandera as pa

schema = pa.DataFrameSchema(
    columns={
        "col1": pa.Column(pa.Int, pa.Check(lambda s: s >= 0)),
        "col2": pa.Column(pa.String, pa.Check(lambda x: x <= 0)),
        "col3": pa.Column(pa.Object, pa.Check(lambda x: x == 0)),
    },
    strict=True,
)

new_schema = schema.remove_columns(["col2", "col3"])
print(new_schema)
```

```
DataFrameSchema(
    columns={
```

(continues on next page)

(continued from previous page)

```
        "col1": "<Schema Column: 'col1' type=int>"
    },
    checks=[],
    index=None,
    transformer=None,
    coerce=False,
    strict=True
)
```

5.2 Series Schemas

The *SeriesSchema* class allows for the validation of pandas `Series` objects, and are very similar to *columns* and *indexes* described in *DataFrameSchemas*.

```
import pandas as pd
import pandera as pa


# specify multiple validators
schema = pa.SeriesSchema(
    pa.String,
    checks=[
        pa.Check(lambda s: s.str.startswith("foo")),
        pa.Check(lambda s: s.str.endswith("bar")),
        pa.Check(lambda x: len(x) > 3, element_wise=True)
    ],
    nullable=False,
    allow_duplicates=True,
    name="my_series")

validated_series = schema.validate(
    pd.Series(["foobar", "foobar", "foobar"], name="my_series"))
print(validated_series)
```

```
0    foobar
1    foobar
2    foobar
Name: my_series, dtype: object
```

5.3 Checks

5.3.1 Checking column properties

Check objects accept a function as a required argument, which is expected to take a `pa.Series` input and output a `boolean` or a `Series` of boolean values. For the check to pass, all of the elements in the boolean series must evaluate to `True`, for example:

```
import pandera as pa

check_lt_10 = pa.Check(lambda s: s <= 10)
```

(continues on next page)

(continued from previous page)

```
schema = pa.DataFrameSchema({"column1": pa.Column(pa.Int, check_lt_10)})
schema.validate(pd.DataFrame({"column1": range(10)}))
```

Multiple checks can be applied to a column:

```
schema = pa.DataFrameSchema({
    "column2": pa.Column(pa.String, [
        pa.Check(lambda s: s.str.startswith("value")),
        pa.Check(lambda s: s.str.split("_", expand=True).shape[1] == 2)
    ]),
})
```

5.3.2 Built-in Checks

For common validation tasks, built-in checks are available in `pandera`.

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

schema = DataFrameSchema({
    "small_values": Column(pa.Float, Check.less_than(100)),
    "one_to_three": Column(pa.Int, Check.isin([1, 2, 3])),
    "phone_number": Column(pa.String, Check.str_matches(r'^[a-z0-9-]+$')),
})
```

See the *Check* API reference for a complete list of built-in checks.

5.3.3 Vectorized vs. Element-wise Checks

By default, *Check* objects operate on `pd.Series` objects. If you want to make atomic checks for each element in the Column, then you can provide the `element_wise=True` keyword argument:

```
import pandas as pd
import pandera as pa

schema = pa.DataFrameSchema({
    "a": pa.Column(
        pa.Int,
        checks=[
            # a vectorized check that returns a bool
            pa.Check(lambda s: s.mean() > 5, element_wise=False),

            # a vectorized check that returns a boolean series
            pa.Check(lambda s: s > 0, element_wise=False),

            # an element-wise check that returns a bool
            pa.Check(lambda x: x > 0, element_wise=True),
        ]
    ),
})
df = pd.DataFrame({"a": [4, 4, 5, 6, 6, 7, 8, 9]})
schema.validate(df)
```

`element_wise == False` by default so that you can take advantage of the speed gains provided by the `pd.Series` API by writing vectorized checks.

5.3.4 Handling Null Values

By default, `pandera` drops null values before passing the objects to validate into the check function. For `Series` objects null elements are dropped (this also applies to columns), and for `DataFrame` objects, rows with any null value are dropped.

If you want to check the properties of a pandas data structure while preserving null values, specify `Check(..., ignore_na=False)` when defining a check.

Note that this is different from the `nullable` argument in *Column* objects, which simply checks for null values in a column.

5.3.5 Column Check Groups

Column checks support grouping by a different column so that you can make assertions about subsets of the column of interest. This changes the function signature of the *Check* function so that its input is a dict where keys are the group names and values are subsets of the series being validated.

Specifying `groupby` as a column name, list of column names, or callable changes the expected signature of the *Check* function argument to:

`Callable[Dict[Any, pd.Series] -> Union[bool, pd.Series]`

where the dict keys are the discrete keys in the `groupby` columns.

In the example below we define a *DataFrameSchema* with column checks for `height_in_feet` using a single column, multiple columns, and a more complex groupby function that creates a new column `age_less_than_15` on the fly.

```
import pandas as pd
import pandera as pa

schema = pa.DataFrameSchema({
    "height_in_feet": pa.Column(
        pa.Float, [
            # groupby as a single column
            pa.Check(
                lambda g: g[False].mean() > 6,
                groupby="age_less_than_20"),

            # define multiple groupby columns
            pa.Check(
                lambda g: g[(True, "F")].sum() == 9.1,
                groupby=["age_less_than_20", "sex"]),

            # groupby as a callable with signature:
            # (DataFrame) -> DataFrameGroupBy
            pa.Check(
                lambda g: g[(False, "M")].median() == 6.75,
                groupby=lambda df: (
                    df.assign(age_less_than_15=lambda d: d["age"] < 15)
                    .groupby(["age_less_than_15", "sex"]))),
        ]),
    "age": pa.Column(pa.Int, pa.Check(lambda s: s > 0)),
```

(continues on next page)

(continued from previous page)

```
    "age_less_than_20": pa.Column(pa.Bool),
    "sex": pa.Column(pa.String, pa.Check(lambda s: s.isin(["M", "F"])))
})

df = (
    pd.DataFrame({
        "height_in_feet": [6.5, 7, 6.1, 5.1, 4],
        "age": [25, 30, 21, 18, 13],
        "sex": ["M", "M", "F", "F", "F"]
    })
    .assign(age_less_than_20=lambda x: x["age"] < 20)
)

schema.validate(df)
```

5.3.6 Wide Checks

`pandera` is primarily designed to operate on long-form data (commonly known as tidy data), where each row is an observation and each column is an attribute associated with an observation.

However, `pandera` also supports checks on wide-form data to operate across columns in a `DataFrame`. For example, if you want to make assertions about `height` across two groups, the tidy dataset and schema might look like this:

```
import pandas as pd
import pandera as pa


df = pd.DataFrame({
    "height": [5.6, 6.4, 4.0, 7.1],
    "group": ["A", "B", "A", "B"],
})

schema = pa.DataFrameSchema({
    "height": pa.Column(
        pa.Float,
        pa.Check(lambda g: g["A"].mean() < g["B"].mean(), groupby="group")
    ),
    "group": pa.Column(pa.String)
})

schema.validate(df)
```

Whereas the equivalent wide-form schema would look like this:

```
df = pd.DataFrame({
    "height_A": [5.6, 4.0],
    "height_B": [6.4, 7.1],
})

schema = pa.DataFrameSchema(
    columns={
        "height_A": pa.Column(pa.Float),
        "height_B": pa.Column(pa.Float),
    },
```

(continues on next page)

(continued from previous page)

```
    # define checks at the DataFrameSchema-level
    checks=pa.Check(
        lambda df: df["height_A"].mean() < df["height_B"].mean()
    )
)

schema.validate(df)
```

You can see that when checks are supplied to the `DataFrameSchema` `checks` key-word argument, the check function should expect a pandas `DataFrame` and should return a `bool`, a `Series` of booleans, or a `DataFrame` of boolean values.

5.3.7 Raise UserWarning on Check Failure

In some cases, you might want to raise a `UserWarning` and continue execution of your program. The `Check` and `Hypothesis` classes and their built-in methods support the keyword argument `raise_warning`, which is `False` by default. If set to `True`, the check will raise a `UserWarning` instead of raising a `SchemaError` exception.

Note: Use this feature carefully! If the check is for informational purposes and not critical for data integrity then use `raise_warning=True`. However, if the assumptions expressed in a `Check` are necessary conditions to considering your data valid, do not set this option to true.

One scenario where you'd want to do this would be in a data pipeline that does some preprocessing, checks for normality in certain columns, and writes the resulting dataset to a table. In this case, you want to see if your normality assumptions are not fulfilled by certain columns, but you still want the resulting table for further analysis.

```
import warnings

import numpy as np
import pandas as pd
import pandera as pa

from scipy.stats import normaltest


np.random.seed(1000)

df = pd.DataFrame({
    "var1": np.random.normal(loc=0, scale=1, size=1000),
    "var2": np.random.uniform(low=0, high=10, size=1000),
})

normal_check = pa.Hypothesis(
    test=normaltest,
    samples="normal_variable",
    # null hypotheses: sample comes from a normal distribution. The
    # relationship function checks if we cannot reject the null hypothesis,
    # i.e. the p-value is greater or equal to alpha.
    relationship=lambda stat, pvalue, alpha=0.05: pvalue >= alpha,
    error="normality test",
    raise_warning=True,
)
```

(continues on next page)

(continued from previous page)

```
schema = pa.DataFrameSchema(
    columns={
        "var1": pa.Column(checks=normal_check),
        "var2": pa.Column(checks=normal_check),
    }
)

# catch and print warnings
with warnings.catch_warnings(record=True) as caught_warnings:
    warnings.simplefilter("always")
    validated_df = schema(df)
    for warning in caught_warnings:
        print(warning.message)
```

```
<Schema Column: 'var2' type=None> failed series validator 0:
<Check _hypothesis_check: normality test>
```

5.4 Hypothesis Testing

`pandera` enables you to perform statistical hypothesis tests on your data.

The *`Hypothesis`* class defines built in methods, which can be called as in this example of a two-sample t-test:

```
import pandas as pd
import pandera as pa

from pandera import Column, DataFrameSchema, Check, Hypothesis

from scipy import stats

df = (
    pd.DataFrame({
        "height_in_feet": [6.5, 7, 6.1, 5.1, 4],
        "sex": ["M", "M", "F", "F", "F"]
    })
)

schema = DataFrameSchema({
    "height_in_feet": Column(
        pa.Float, [
            Hypothesis.two_sample_ttest(
                sample1="M",
                sample2="F",
                groupby="sex",
                relationship="greater_than",
                alpha=0.05,
                equal_var=True),
    ]),
    "sex": Column(pa.String)
})

schema.validate(df)
```

```
Traceback (most recent call last):
...
pandera.SchemaError: <Schema Column: 'height_in_feet' type=float64> failed series␣
↪validator 0: hypothesis_check: failed two sample ttest between 'M' and 'F'
```

You can also define custom hypotheses by passing in functions to the `test` and `relationship` arguments.

The `test` function takes as input one or multiple array-like objects and should return a `stat`, which is the test statistic, and `pvalue` for assessing statistical significance. It also takes key-word arguments supplied by the `test_kwargs` dict when initializing a `Hypothesis` object.

The `relationship` function should take all of the outputs of `test` as positional arguments, in addition to key-word arguments supplied by the `relationship_kwargs` dict.

Here's an implementation of the two-sample t-test that uses the scipy implementation:

```
def two_sample_ttest(array1, array2):
    # the "height_in_feet" series is first grouped by "sex" and then
    # passed into the custom `test` function as two separate arrays in the
    # order specified in the `samples` argument.
    return stats.ttest_ind(array1, array2)


def null_relationship(stat, pvalue, alpha=0.01):
    return pvalue / 2 >= alpha


schema = DataFrameSchema({
    "height_in_feet": Column(
        pa.Float, [
            Hypothesis(
                test=two_sample_ttest,
                samples=["M", "F"],
                groupby="sex",
                relationship=null_relationship,
                relationship_kwargs={"alpha": 0.05}
            )
    ]),
    "sex": Column(pa.String, checks=Check.isin(["M", "F"]))
})

schema.validate(df)
```

5.4.1 Wide Hypotheses

`pandera` is primarily designed to operate on long-form data (commonly known as tidy data), where each row is an observation and columns are attributes associated with the observation.

However, `pandera` also supports hypothesis testing on wide-form data to operate across columns in a `DataFrame`.

For example, if you want to make assertions about `height` across two groups, the tidy dataset and schema might look like this:

```
import pandas as pd
import pandera as pa

from pandera import Check, DataFrameSchema, Column, Hypothesis
```

(continues on next page)

(continued from previous page)

```
df = pd.DataFrame({
    "height": [5.6, 7.5, 4.0, 7.9],
    "group": ["A", "B", "A", "B"],
})

schema = DataFrameSchema({
    "height": Column(
        pa.Float, Hypothesis.two_sample_ttest(
            "A", "B",
            groupby="group",
            relationship="less_than",
            alpha=0.05
        )
    ),
    "group": Column(pa.String, Check(lambda s: s.isin(["A", "B"])))
})

schema.validate(df)
```

The equivalent wide-form schema would look like this:

```
import pandas as pd
import pandera as pa

from pandera import DataFrameSchema, Column, Hypothesis

df = pd.DataFrame({
    "height_A": [5.6, 4.0],
    "height_B": [7.5, 7.9],
})

schema = DataFrameSchema(
    columns={
        "height_A": Column(Float),
        "height_B": Column(Float),
    },
    # define checks at the DataFrameSchema-level
    checks=Hypothesis.two_sample_ttest(
        "height_A", "height_B",
        relationship="less_than",
        alpha=0.05
    )
)

schema.validate(df)
```

5.5 Decorators for Pipeline Integration

If you have an existing data pipeline that uses pandas data structures, you can use the *check_input()* and *check_output()* decorators to easily check function arguments or returned variables from existing functions.

5.5.1 Check Input

Validates input pandas DataFrame/Series before entering the wrapped function.

```
import pandas as pd
import pandera as pa

from pandera import DataFrameSchema, Column, Check, check_input


df = pd.DataFrame({
   "column1": [1, 4, 0, 10, 9],
   "column2": [-1.3, -1.4, -2.9, -10.1, -20.4],
})

in_schema = DataFrameSchema({
   "column1": Column(pa.Int,
                     Check(lambda x: 0 <= x <= 10, element_wise=True)),
   "column2": Column(pa.Float, Check(lambda x: x < -1.2)),
})

# by default, check_input assumes that the first argument is
# dataframe/series.
@check_input(in_schema)
def preprocessor(dataframe):
    dataframe["column3"] = dataframe["column1"] + dataframe["column2"]
    return dataframe

preprocessed_df = preprocessor(df)
print(preprocessed_df)
```

```
   column1  column2  column3
0        1     -1.3     -0.3
1        4     -1.4      2.6
2        0     -2.9     -2.9
3       10    -10.1     -0.1
4        9    -20.4    -11.4
```

You can also provide the argument name as a string

```
@check_input(in_schema, "dataframe")
def preprocessor(dataframe):
    ...
```

Or an integer representing the index in the positional arguments.

```
@check_input(in_schema, 1)
def preprocessor(foo, dataframe):
    ...
```

5.5.2 Check Output

The same as `check_input`, but this decorator checks the output DataFrame/Series of the decorated function.

```
import pandas as pd
import pandera as pa

from pandera import DataFrameSchema, Column, Check, check_output


preprocessed_df = pd.DataFrame({
   "column1": [1, 4, 0, 10, 9],
})

# assert that all elements in "column1" are zero
out_schema = DataFrameSchema({
    "column1": Column(pa.Int, Check(lambda x: x == 0))
})


# by default assumes that the pandas DataFrame/Schema is the only output
@check_output(out_schema)
def zero_column_1(df):
    df["column1"] = 0
    return df


# you can also specify in the index of the argument if the output is list-like
@check_output(out_schema, 1)
def zero_column_1_arg(df):
    df["column1"] = 0
    return "foobar", df


# or the key containing the data structure to verify if the output is dict-like
@check_output(out_schema, "out_df")
def zero_column_1_dict(df):
    df["column1"] = 0
    return {"out_df": df, "out_str": "foobar"}


# for more complex outputs, you can specify a function
@check_output(out_schema, lambda x: x[1]["out_df"])
def zero_column_1_custom(df):
    df["column1"] = 0
    return ("foobar", {"out_df": df})


zero_column_1(preprocessed_df)
zero_column_1_arg(preprocessed_df)
zero_column_1_dict(preprocessed_df)
zero_column_1_custom(preprocessed_df)
```

5.6 Schema Inference

New in version 0.4.0

Warning: This functionality is experimental. Use with caution!

With simple use cases, writing a schema definition manually is pretty straight-forward with pandera. However, it can get tedious to do this with dataframes that have many columns of various data types.

To help you handle these cases, the *infer_schema()* function enables you to quickly infer a draft schema from a pandas dataframe or series. Below is a simple example:

```
import pandas as pd
import pandera as pa

from pandera import Check, Column, DataFrameSchema

df = pd.DataFrame({
    "column1": [5, 10, 20],
    "column2": ["a", "b", "c"],
    "column3": pd.to_datetime(["2010", "2011", "2012"]),
})
schema = pa.infer_schema(df)
print(schema)
```

```
DataFrameSchema(
    columns={
        "column1": "<Schema Column: 'column1' type=int64>",
        "column2": "<Schema Column: 'column2' type=string>",
        "column3": "<Schema Column: 'column3' type=datetime64[ns]>"
    },
    checks=[],
    index=<Schema Index>,
    transformer=None,
    coerce=True,
    strict=False
)
```

These inferred schemas are **rough drafts** that shouldn't be used for validation without modification. You can modify the inferred schema to obtain the schema definition that you're satisfied with.

For *DataFrameSchema* objects, the following methods create modified copies of the schema:

- *DataFrameSchema.add_columns()*
- *DataFrameSchema.remove_columns()*
- *DataFrameSchema.update_column()*

For *SeriesSchema* objects:

- SeriesSchema.set_checks()

The section below describes two workflows for persisting and modifying an inferred schema.

5.6.1 Schema Persistence

There are two ways of persisting schemas, inferred or otherwise.

5.6.1.1 Write to a Python script

You can also write your schema to a python script with *io.to_script()*:

```
from pandera import io

# supply a file-like object, Path, or str to write to a file. If not
# specified, to_script will output the code as a string.
schema_script = io.to_script(schema)
print(schema_script)
```

```
from pandas import Timestamp
from pandera import (
    DataFrameSchema,
    Column,
    Check,
    Index,
    MultiIndex,
    PandasDtype,
)

schema = DataFrameSchema(
    columns={
        "column1": Column(
            pandas_dtype=PandasDtype.Int64,
            checks=[
                Check.greater_than_or_equal_to(min_value=5.0),
                Check.less_than_or_equal_to(max_value=20.0),
            ],
            nullable=False,
            allow_duplicates=True,
            coerce=False,
            required=True,
            regex=False,
        ),
        "column2": Column(
            pandas_dtype=PandasDtype.String,
            checks=None,
            nullable=False,
            allow_duplicates=True,
            coerce=False,
            required=True,
            regex=False,
        ),
        "column3": Column(
            pandas_dtype=PandasDtype.DateTime,
            checks=[
                Check.greater_than_or_equal_to(
                    min_value=Timestamp("2010-01-01 00:00:00")
                ),
                Check.less_than_or_equal_to(
                    max_value=Timestamp("2012-01-01 00:00:00")
```

(continues on next page)

(continued from previous page)

```
                ),
            ],
            nullable=False,
            allow_duplicates=True,
            coerce=False,
            required=True,
            regex=False,
        ),
    },
    index=Index(
        pandas_dtype=PandasDtype.Int64,
        checks=[
            Check.greater_than_or_equal_to(min_value=0.0),
            Check.less_than_or_equal_to(max_value=2.0),
        ],
        nullable=False,
        coerce=False,
        name=None,
    ),
    coerce=True,
    strict=False,
    name=None,
)
```

As a python script, you can iterate on an inferred schema and use it to validate data once you are satisfied with your schema definition.

5.6.1.2 Write to YAML

You can also write the schema object to a yaml file with *io.to_yaml()*, and you can then read it into memory with *io.from_yaml()*. The *DataFrameSchema.to_yaml()* and *DataFrameSchema.from_yaml()* is a convenience method for this functionality.

```
# supply a file-like object, Path, or str to write to a file. If not
# specified, to_yaml will output a yaml string.
yaml_schema = schema.to_yaml()
print(yaml_schema)
```

```
schema_type: dataframe
version: 0.4.5
columns:
  column1:
    pandas_dtype: int64
    nullable: false
    checks:
      greater_than_or_equal_to: 5.0
      less_than_or_equal_to: 20.0
    allow_duplicates: true
    coerce: false
    required: true
    regex: false
  column2:
    pandas_dtype: string
    nullable: false
    checks: null
```

(continues on next page)

(continued from previous page)

```
    allow_duplicates: true
    coerce: false
    required: true
    regex: false
  column3:
    pandas_dtype: datetime64[ns]
    nullable: false
    checks:
      greater_than_or_equal_to: '2010-01-01 00:00:00'
      less_than_or_equal_to: '2012-01-01 00:00:00'
    allow_duplicates: true
    coerce: false
    required: true
    regex: false
index:
- pandas_dtype: int64
  nullable: false
  checks:
    greater_than_or_equal_to: 0.0
    less_than_or_equal_to: 2.0
  name: null
  coerce: false
coerce: true
strict: false
```

You can edit this yaml file by specifying column names under the `column` key. The respective values map onto key-word arguments in the *Column* class.

Note: Currently, only built-in *Check* methods are supported under the `checks` key.

5.7 Lazy Validation

New in version 0.4.0

By default, when you call the `validate` method on schema or schema component objects, a *errors.SchemaError* is raised as soon as one of the assumptions specified in the schema is falsified. For example, for a *DataFrameSchema* object, the following situations will raise an exception:

- a column specified in the schema is not present in the dataframe.
- if `strict=True`, a column in the dataframe is not specified in the schema.
- the `pandas_dtype` does not match.
- if `coerce=True`, the dataframe column cannot be coerced into the specified `pandas_dtype`.
- the *Check* specified in one of the columns returns `False` or a boolean series containing at least one `False` value.

For example:

```
import pandas as pd
import pandera as pa

from pandera import Check, Column, DataFrameSchema
```

(continues on next page)

(continued from previous page)

```
df = pd.DataFrame({"column": ["a", "b", "c"]})

schema = pa.DataFrameSchema({"column": Column(pa.Int)})
schema.validate(df)
```

```
Traceback (most recent call last):
...
SchemaError: expected series 'column' to have type int64, got object
```

For more complex cases, it is useful to see all of the errors raised during the `validate` call so that you can debug the causes of errors on different columns and checks. The `lazy` keyword argument in the `validate` method of all schemas and schema components gives you the option of doing just this:

```
import pandas as pd
import pandera as pa

from pandera import Check, Column, DataFrameSchema

schema = pa.DataFrameSchema(
    columns={
        "int_column": Column(pa.Int),
        "float_column": Column(pa.Float, Check.greater_than(0)),
        "str_column": Column(pa.String, Check.equal_to("a")),
        "date_column": Column(pa.DateTime),
    },
    strict=True
)

df = pd.DataFrame({
    "int_column": ["a", "b", "c"],
    "float_column": [0, 1, 2],
    "str_column": ["a", "b", "d"],
    "unknown_column": None,
})

schema.validate(df, lazy=True)
```

```
Traceback (most recent call last):
...
pandera.errors.SchemaErrors: A total of 5 schema errors were found.

Error Counts
------------
- column_not_in_schema: 1
- column_not_in_dataframe: 1
- schema_component_check: 3

Schema Error Summary
--------------------
                                                        failure_cases  n_failure_
↪cases
schema_context  column       check
DataFrameSchema <NA>         column_in_dataframe         [date_column]             ␣
↪1
                             column_in_schema         [unknown_column]             ␣
↪1
```

(continues on next page)

(continued from previous page)

```
Column           float_column pandas_dtype('float64')            [int64]                ␣
↪1
                 int_column   pandas_dtype('int64')             [object]               ␣
↪1
                 str_column   equal_to(a)                         [b, d]               ␣
↪2

Usage Tip
---------

Directly inspect all errors by catching the exception:

```
try:
    schema.validate(dataframe, lazy=True)
except SchemaErrors as err:
    err.schema_errors # dataframe of schema errors
    err.data # invalid dataframe
```
```

As you can see from the output above, a *errors.SchemaErrors* exception is raised with a summary of the error counts and failure cases caught by the schema. You can also see from the **Usage Tip** that you can catch these errors and inspect the failure cases in a more granular form:

```
try:
    schema.validate(df, lazy=True)
except pa.errors.SchemaErrors as err:
    print("Schema errors and failure cases:")
    print(err.schema_errors.head())
    print("\nDataFrame object that failed validation:")
    print(err.data.head())
```

```
Schema errors and failure cases:
    schema_context        column                    check check_number  \
0  DataFrameSchema          None         column_in_schema         None
1  DataFrameSchema          None      column_in_dataframe         None
2           Column    int_column    pandas_dtype('int64')         None
3           Column  float_column  pandas_dtype('float64')         None
4           Column    str_column              equal_to(a)            0

     failure_case index
0  unknown_column  None
1     date_column  None
2          object  None
3           int64  None
4               b     1

DataFrame object that failed validation:
  int_column  float_column str_column unknown_column
0          a             0          a           None
1          b             1          b           None
2          c             2          d           None
```

5.8 API Reference

5.8.1 Schemas

DataFrameSchema	A light-weight pandas DataFrame validator.
SeriesSchema	Series validator.

5.8.1.1 pandera.DataFrameSchema

class pandera.**DataFrameSchema**(*columns=None*, *checks=None*, *index=None*, *transformer=None*, *coerce=False*, *strict=False*, *name=None*)

A light-weight pandas DataFrame validator.

Initialize DataFrameSchema validator.

Parameters

- **columns** (*mapping of column names and column schema component.*) – a dict where keys are column names and values are Column objects specifying the datatypes and properties of a particular column.
- **checks** (Union[Check, Hypothesis, List[Union[Check, Hypothesis]], None]) – dataframe-wide checks.
- **index** – specify the datatypes and properties of the index.
- **transformer** (Optional[Callable]) – a callable with signature: pandas.DataFrame -> pandas.DataFrame. If specified, calling *validate* will verify properties of the columns and return the transformed dataframe object.
- **coerce** (bool) – whether or not to coerce all of the columns on validation.
- **strict** – whether or not to accept columns in the dataframe that aren't in the DataFrameSchema.
- **name** (Optional[str]) – name of the schema.

Raises ***SchemaInitError*** – if impossible to build schema from parameters

Examples

```
>>> import pandera as pa
>>>
>>> schema = pa.DataFrameSchema({
...     "str_column": pa.Column(pa.String),
...     "float_column": pa.Column(pa.Float),
...     "int_column": pa.Column(pa.Int),
...     "date_column": pa.Column(pa.DateTime),
... })
```

Use the pandas API to define checks, which takes a function with the signature: `pd.Series -> Union[bool, pd.Series]` where the output series contains boolean values.

```
>>> from pandera import Check
>>>
>>> schema_withchecks = pa.DataFrameSchema({
...     "probability": pa.Column(
...         pa.Float, pa.Check(lambda s: (s >= 0) & (s <= 1))),
```

(continues on next page)

(continued from previous page)

```
...
...     # check that the "category" column contains a few discrete
...     # values, and the majority of the entries are dogs.
...     "category": pa.Column(
...         pa.String, [
...             pa.Check(lambda s: s.isin(["dog", "cat", "duck"])),
...             pa.Check(lambda s: (s == "dog").mean() > 0.5),
...         ]),
... })
```

See *here* for more usage details.

Attributes

coerce	Whether to coerce series to specified type.
dtype	A pandas style dtype dict where the keys are column names and values are pandas dtype for the column.

Methods

__init__	Initialize DataFrameSchema validator.
add_columns	Create a copy of the DataFrameSchema with extra columns.
from_yaml	Create DataFrameSchema from yaml file.
get_dtype	Same as the dtype property, but expands columns where regex == True based on the supplied dataframe.
remove_columns	Removes columns from a DataFrameSchema and returns a new copy.
rename_columns	Rename columns using a dictionary of key-value pairs.
select_columns	Select subset of columns in the schema.
to_yaml	Write DataFrameSchema to yaml file.
update_column	Create copy of a DataFrameSchema with updated column properties.
validate	Check if all columns in a dataframe have a column in the Schema.
__call__	Alias for *DataFrameSchema.validate()* method.

5.8.1.1.1 pandera.DataFrameSchema.__init__

DataFrameSchema.**__init__**(*columns=None*, *checks=None*, *index=None*, *transformer=None*, *coerce=False*, *strict=False*, *name=None*)

Initialize DataFrameSchema validator.

Parameters

- **columns** (*mapping of column names and column schema component.*) – a dict where keys are column names and values are Column objects specifying the datatypes and properties of a particular column.
- **checks** (Union[Check, Hypothesis, List[Union[Check, Hypothesis]], None]) – dataframe-wide checks.
- **index** – specify the datatypes and properties of the index.
- **transformer** (Optional[Callable]) – a callable with signature: pandas.DataFrame -> pandas.DataFrame. If specified, calling *validate* will verify properties of the columns and return the transformed dataframe object.
- **coerce** (bool) – whether or not to coerce all of the columns on validation.
- **strict** – whether or not to accept columns in the dataframe that aren't in the DataFrameSchema.
- **name** (Optional[str]) – name of the schema.

Raises ***SchemaInitError*** – if impossible to build schema from parameters

Examples

```
>>> import pandera as pa
>>>
>>> schema = pa.DataFrameSchema({
...     "str_column": pa.Column(pa.String),
...     "float_column": pa.Column(pa.Float),
...     "int_column": pa.Column(pa.Int),
...     "date_column": pa.Column(pa.DateTime),
... })
```

Use the pandas API to define checks, which takes a function with the signature: pd.Series -> Union[bool, pd.Series] where the output series contains boolean values.

```
>>> from pandera import Check
>>>
>>> schema_withchecks = pa.DataFrameSchema({
...     "probability": pa.Column(
...         pa.Float, pa.Check(lambda s: (s >= 0) & (s <= 1))),
...
...     # check that the "category" column contains a few discrete
...     # values, and the majority of the entries are dogs.
...     "category": pa.Column(
...         pa.String, [
...             pa.Check(lambda s: s.isin(["dog", "cat", "duck"])),
...             pa.Check(lambda s: (s == "dog").mean() > 0.5),
...         ]),
... })
```

See *here* for more usage details.

Return type None

5.8.1.1.2 pandera.DataFrameSchema.add_columns

DataFrameSchema.**add_columns**(*extra_schema_cols*)

Create a copy of the DataFrameSchema with extra columns.

Parameters **extra_schema_cols** (DataFrameSchema) – Additional columns of the format

Return type DataFrameSchema

Returns a new DataFrameSchema with the extra_schema_cols added

5.8.1.1.3 pandera.DataFrameSchema.from_yaml

classmethod DataFrameSchema.**from_yaml**(*yaml_schema*)

Create DataFrameSchema from yaml file.

Parameters **yaml_schema** – str, Path to yaml schema, or serialized yaml string.

Return type DataFrameSchema

Returns dataframe schema.

5.8.1.1.4 pandera.DataFrameSchema.get_dtype

DataFrameSchema.**get_dtype**(*dataframe*)

Same as the dtype property, but expands columns where regex == True based on the supplied dataframe.

Return type Dict[str, str]

Returns dictionary of columns and their associated dtypes.

5.8.1.1.5 pandera.DataFrameSchema.remove_columns

DataFrameSchema.**remove_columns**(*cols_to_remove*)

Removes columns from a DataFrameSchema and returns a new copy.

Parameters **cols_to_remove** (*List*) – Columns to be removed from the DataFrameSchema

Return type DataFrameSchema

Returns a new DataFrameSchema without the cols_to_remove

5.8.1.1.6 pandera.DataFrameSchema.rename_columns

DataFrameSchema.**rename_columns**(*rename_dict*)

Rename columns using a dictionary of key-value pairs.

Parameters **rename_dict** (dict) – dictionary of 'old_name': 'new_name' key-value pairs.

Returns dataframe schema (copy of original)

5.8.1.1.7 pandera.DataFrameSchema.select_columns

DataFrameSchema.**select_columns**(*columns*)

Select subset of columns in the schema.

New in version 0.4.5

Parameters **columns** (list) – list of column names to select.

Returns dataframe schema (copy of original)

5.8.1.1.8 pandera.DataFrameSchema.to_yaml

DataFrameSchema.**to_yaml**(*fp=None*)

Write DataFrameSchema to yaml file.

Parameters

- **dataframe_schema** – schema to write to file or dump to string.
- **stream** – file stream to write to. If None, dumps to string.

Returns yaml string if stream is None, otherwise returns None.

5.8.1.1.9 pandera.DataFrameSchema.update_column

DataFrameSchema.**update_column**(*column_name*, ***kwargs*)

Create copy of a DataFrameSchema with updated column properties.

Parameters

- **column_name** (str) –
- **kwargs** – key-word arguments supplied to *Column*

Return type DataFrameSchema

Returns a new DataFrameSchema with updated column

5.8.1.1.10 pandera.DataFrameSchema.validate

DataFrameSchema.**validate**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*)

Check if all columns in a dataframe have a column in the Schema.

Parameters

- **dataframe** (*pd.DataFrame*) – the dataframe to be validated.
- **head** (Optional[int]) – validate the first n rows. Rows overlapping with *tail* or *sample* are de-duplicated.
- **tail** (Optional[int]) – validate the last n rows. Rows overlapping with *head* or *sample* are de-duplicated.
- **sample** (Optional[int]) – validate a random sample of n rows. Rows overlapping with *head* or *tail* are de-duplicated.
- **random_state** (Optional[int]) – random seed for the sample argument.

- **lazy** (bool) – if True, lazily evaluates dataframe against all validation checks and raises a SchemaErrorReport. Otherwise, raise SchemaError as soon as one occurs.

Return type DataFrame

Returns validated DataFrame

Raises ***SchemaError*** – when DataFrame violates built-in or custom checks.

Example

Calling schema.validate returns the dataframe.

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>> df = pd.DataFrame({
...     "probability": [0.1, 0.4, 0.52, 0.23, 0.8, 0.76],
...     "category": ["dog", "dog", "cat", "duck", "dog", "dog"]
... })
>>>
>>> schema_withchecks = pa.DataFrameSchema({
...     "probability": pa.Column(
...         pa.Float, pa.Check(lambda s: (s >= 0) & (s <= 1))),
...
...     # check that the "category" column contains a few discrete
...     # values, and the majority of the entries are dogs.
...     "category": pa.Column(
...         pa.String, [
...             pa.Check(lambda s: s.isin(["dog", "cat", "duck"])),
...             pa.Check(lambda s: (s == "dog").mean() > 0.5),
...         ]),
... })
>>>
>>> schema_withchecks.validate(df)[["probability", "category"]]
   probability category
0         0.10      dog
1         0.40      dog
2         0.52      cat
3         0.23     duck
4         0.80      dog
5         0.76      dog
```

5.8.1.1.11 pandera.DataFrameSchema.__call__

DataFrameSchema.**__call__**(*dataframe*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*)

Alias for *DataFrameSchema.validate()* method.

Parameters

- **dataframe** (*pd.DataFrame*) – the dataframe to be validated.
- **head** (*int*) – validate the first n rows. Rows overlapping with *tail* or *sample* are de-duplicated.
- **tail** (*int*) – validate the last n rows. Rows overlapping with *head* or *sample* are de-duplicated.

- **sample** (Optional[int]) – validate a random sample of n rows. Rows overlapping with *head* or *tail* are de-duplicated.
- **random_state** (Optional[int]) – random seed for the sample argument.
- **lazy** (bool) – if True, lazily evaluates dataframe against all validation checks and raises a SchemaErrorReport. Otherwise, raise SchemaError as soon as one occurs.

5.8.1.2 pandera.SeriesSchema

class pandera.**SeriesSchema**(*pandas_dtype=None*, *checks=None*, *index=None*, *nullable=False*, *allow_duplicates=True*, *coerce=False*, *name=None*)

Series validator.

Initialize series schema base object.

Parameters

- **pandas_dtype** (Union[str, type, PandasDtype, ExtensionDtype, None]) – datatype of the column. If a string is specified, then assumes one of the valid pandas string values: http://pandas.pydata.org/pandas-docs/stable/basics.html#dtypes
- **checks** (*callable*) – If element_wise is True, then callable signature should be:

  Callable[Any, bool] where the Any input is a scalar element in the column. Otherwise, the input is assumed to be a pandas.Series object.
- **index** – specify the datatypes and properties of the index.
- **nullable** (*bool*) – Whether or not column can contain null values.
- **allow_duplicates** (*bool*) –

Attributes

allow_duplicates	Whether to allow duplicate values.
checks	Return list of checks or hypotheses.
coerce	Whether to coerce series to specified type.
dtype	String representation of the dtype.
name	Get SeriesSchema name.
nullable	Whether the series is nullable.
pandas_dtype	Get the pandas dtype

Methods

__init__	Initialize series schema base object.
validate	Validate a Series object.
__call__	Alias for *SeriesSchema.validate()* method.

5.8.1.2.1 pandera.SeriesSchema.__init__

SeriesSchema.**__init__**(*pandas_dtype=None*, *checks=None*, *index=None*, *nullable=False*, *allow_duplicates=True*, *coerce=False*, *name=None*)

Initialize series schema base object.

Parameters

- **pandas_dtype** (Union[str, type, PandasDtype, ExtensionDtype, None]) – datatype of the column. If a string is specified, then assumes one of the valid pandas string values: http://pandas.pydata.org/pandas-docs/stable/basics.html#dtypes
- **checks** (*callable*) – If element_wise is True, then callable signature should be:

  Callable[Any, bool] where the Any input is a scalar element in the column. Otherwise, the input is assumed to be a pandas.Series object.
- **index** – specify the datatypes and properties of the index.
- **nullable** (*bool*) – Whether or not column can contain null values.
- **allow_duplicates** (*bool*) –

Return type None

5.8.1.2.2 pandera.SeriesSchema.validate

SeriesSchema.**validate**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*)

Validate a Series object.

Parameters

- **check_obj** (Series) – One-dimensional ndarray with axis labels (including time series).
- **head** (Optional[int]) – validate the first n rows. Rows overlapping with *tail* or *sample* are de-duplicated.
- **tail** (Optional[int]) – validate the last n rows. Rows overlapping with *head* or *sample* are de-duplicated.
- **sample** (Optional[int]) – validate a random sample of n rows. Rows overlapping with *head* or *tail* are de-duplicated.
- **random_state** (Optional[int]) – random seed for the sample argument.
- **lazy** (bool) – if True, lazily evaluates dataframe against all validation checks and raises a SchemaErrorReport. Otherwise, raise SchemaError as soon as one occurs.

Return type Series

Returns validated Series.

Raises ***SchemaError*** – when DataFrame violates built-in or custom checks.

Example

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>> series_schema = pa.SeriesSchema(
```

(continues on next page)

(continued from previous page)

```
...     pa.Float, [
...         pa.Check(lambda s: s > 0),
...         pa.Check(lambda s: s < 1000),
...         pa.Check(lambda s: s.mean() > 300),
...     ])
>>> series = pd.Series([1, 100, 800, 900, 999], dtype=float)
>>> print(series_schema.validate(series))
0      1.0
1    100.0
2    800.0
3    900.0
4    999.0
dtype: float64
```

5.8.1.2.3 pandera.SeriesSchema.__call__

SeriesSchema.**__call__**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*)

Alias for *SeriesSchema.validate()* method.

Return type Series

5.8.2 Schema Components

Column	Validate types and properties of DataFrame columns.
Index	Validate types and properties of a DataFrame Index.
MultiIndex	Validate types and properties of a DataFrame MultiIndex.

5.8.2.1 pandera.Column

class pandera.**Column**(*pandas_dtype=None*, *checks=None*, *nullable=False*, *allow_duplicates=True*, *coerce=False*, *required=True*, *name=None*, *regex=False*)

Validate types and properties of DataFrame columns.

Create column validator object.

Parameters

- **pandas_dtype** (Union[str, type, PandasDtype, ExtensionDtype, None]) – datatype of the column. A PandasDtype for type-checking dataframe. If a string is specified, then assumes one of the valid pandas string values: http://pandas.pydata.org/pandas-docs/stable/basics.html#dtypes
- **checks** (Union[Check, Hypothesis, List[Union[Check, Hypothesis]], None]) – checks to verify validity of the column
- **nullable** (bool) – Whether or not column can contain null values.
- **allow_duplicates** (bool) – Whether or not to coerce the column to the specified pandas_dtype before validation
- **coerce** (bool) – If True, when schema.validate is called the column will be coerced into the specified dtype.

- **required** (bool) – Whether or not column is allowed to be missing
- **name** (Optional[str]) – column name in dataframe to validate.
- **regex** (bool) – whether the name attribute should be treated as a regex pattern to apply to multiple columns in a dataframe.

Raises ***SchemaInitError*** – if impossible to build schema from parameters

Example

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>>
>>> schema = pa.DataFrameSchema({
...     "column": pa.Column(pa.String)
... })
>>>
>>> schema.validate(pd.DataFrame({"column": ["foo", "bar"]}))
  column
0    foo
1    bar
```

See *here* for more usage details.

Attributes

allow_duplicates	Whether to allow duplicate values.
checks	Return list of checks or hypotheses.
coerce	Whether to coerce series to specified type.
dtype	String representation of the dtype.
name	Get SeriesSchema name.
nullable	Whether the series is nullable.
pandas_dtype	Get the pandas dtype
properties	Get column properties.
regex	True if name attribute should be treated as a regex pattern.

Methods

__init__	Create column validator object.
get_regex_columns	Get matching column names based on regex column name pattern.
set_name	Used to set or modify the name of a column object.
validate	Validate a Column in a DataFrame object.
__call__	Alias for validate method.

5.8.2.1.1 pandera.Column.__init__

Column.**__init__**(*pandas_dtype=None*, *checks=None*, *nullable=False*, *allow_duplicates=True*, *coerce=False*, *required=True*, *name=None*, *regex=False*)

Create column validator object.

Parameters

- **pandas_dtype** (Union[str, type, PandasDtype, ExtensionDtype, None]) – datatype of the column. A PandasDtype for type-checking dataframe. If a string is specified, then assumes one of the valid pandas string values: http://pandas.pydata.org/pandas-docs/stable/basics.html#dtypes
- **checks** (Union[Check, Hypothesis, List[Union[Check, Hypothesis]], None]) – checks to verify validity of the column
- **nullable** (bool) – Whether or not column can contain null values.
- **allow_duplicates** (bool) – Whether or not to coerce the column to the specified pandas_dtype before validation
- **coerce** (bool) – If True, when schema.validate is called the column will be coerced into the specified dtype.
- **required** (bool) – Whether or not column is allowed to be missing
- **name** (Optional[str]) – column name in dataframe to validate.
- **regex** (bool) – whether the name attribute should be treated as a regex pattern to apply to multiple columns in a dataframe.

Raises ***SchemaInitError*** – if impossible to build schema from parameters

Example

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>>
>>> schema = pa.DataFrameSchema({
...     "column": pa.Column(pa.String)
... })
>>>
>>> schema.validate(pd.DataFrame({"column": ["foo", "bar"]}))
  column
0    foo
1    bar
```

See *here* for more usage details.

Return type None

5.8.2.1.2 pandera.Column.get_regex_columns

Column.**get_regex_columns**(*columns*)

Get matching column names based on regex column name pattern.

Parameters **columns** (Union[Index, MultiIndex]) – columns to regex pattern match

Return type Union[Index, MultiIndex]

Returns matchin columns

5.8.2.1.3 pandera.Column.set_name

Column.**set_name**(*name*)

Used to set or modify the name of a column object.

Parameters **name** (*str*) – the name of the column object

5.8.2.1.4 pandera.Column.validate

Column.**validate**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*)

Validate a Column in a DataFrame object.

Parameters

- **check_obj** (DataFrame) – pandas DataFrame to validate.
- **head** (Optional[int]) – validate the first n rows. Rows overlapping with *tail* or *sample* are de-duplicated.
- **tail** (Optional[int]) – validate the last n rows. Rows overlapping with *head* or *sample* are de-duplicated.
- **sample** (Optional[int]) – validate a random sample of n rows. Rows overlapping with *head* or *tail* are de-duplicated.
- **random_state** (Optional[int]) – random seed for the sample argument.
- **lazy** (bool) – if True, lazily evaluates dataframe against all validation checks and raises a SchemaErrorReport. Otherwise, raise SchemaError as soon as one occurs.

Return type DataFrame

Returns validated DataFrame.

5.8.2.1.5 pandera.Column.__call__

Column.**__call__**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*)

Alias for validate method.

Return type Union[DataFrame, Series]

5.8.2.2 pandera.Index

class pandera.**Index**(*pandas_dtype=None*, *checks=None*, *nullable=False*, *allow_duplicates=True*, *coerce=False*, *name=None*)

Validate types and properties of a DataFrame Index.

Create Index validator.

Parameters

- **pandas_dtype** (Union[str, type, PandasDtype, ExtensionDtype, None]) – datatype of the column. A PandasDtype for type-checking dataframe. If a string is specified, then assumes one of the valid pandas string values: http://pandas.pydata.org/pandas-docs/stable/basics.html#dtypes
- **checks** (Union[Check, Hypothesis, List[Union[Check, Hypothesis]], None]) – checks to verify validity of the index.
- **nullable** (bool) – Whether or not column can contain null values.
- **allow_duplicates** (bool) – Whether or not to coerce the column to the specified pandas_dtype before validation
- **coerce** (bool) – If True, when schema.validate is called the index will be coerced into the specified dtype.
- **name** (Optional[str]) – name of the index

Example

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>>
>>> schema = pa.DataFrameSchema(
...     columns={"column": pa.Column(pa.String)},
...     index=pa.Index(pa.Int, allow_duplicates=False))
>>>
>>> schema.validate(
...     pd.DataFrame({"column": ["foo"] * 3}, index=range(3))
... )
  column
0    foo
1    foo
2    foo
```

See *here* for more usage details.

Attributes

allow_duplicates	Whether to allow duplicate values.
checks	Return list of checks or hypotheses.
coerce	Whether to coerce series to specified type.
dtype	String representation of the dtype.
name	Get SeriesSchema name.
nullable	Whether the series is nullable.
pandas_dtype	Get the pandas dtype

Methods

__init__	Create Index validator.
coerce_dtype	Coerce type of a pd.Index by type specified in pandas_dtype.
validate	Validate DataFrameSchema or SeriesSchema Index.
__call__	Alias for `validate` method.

5.8.2.2.1 pandera.Index.__init__

Index.**__init__**(*pandas_dtype=None*, *checks=None*, *nullable=False*, *allow_duplicates=True*, *coerce=False*, *name=None*)

Create Index validator.

Parameters

- **pandas_dtype** (Union[str, type, PandasDtype, ExtensionDtype, None]) – datatype of the column. A PandasDtype for type-checking dataframe. If a string is specified, then assumes one of the valid pandas string values: http://pandas.pydata.org/pandas-docs/stable/basics.html#dtypes
- **checks** (Union[Check, Hypothesis, List[Union[Check, Hypothesis]], None]) – checks to verify validity of the index.
- **nullable** (bool) – Whether or not column can contain null values.
- **allow_duplicates** (bool) – Whether or not to coerce the column to the specified pandas_dtype before validation
- **coerce** (bool) – If True, when schema.validate is called the index will be coerced into the specified dtype.
- **name** (Optional[str]) – name of the index

Example

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>>
>>> schema = pa.DataFrameSchema(
...     columns={"column": pa.Column(pa.String)},
...     index=pa.Index(pa.Int, allow_duplicates=False))
>>>
>>> schema.validate(
...     pd.DataFrame({"column": ["foo"] * 3}, index=range(3))
... )
  column
0    foo
1    foo
2    foo
```

See *here* for more usage details.

Return type None

5.8.2.2.2 pandera.Index.coerce_dtype

Index.**coerce_dtype**(*series_or_index*)

Coerce type of a pd.Index by type specified in pandas_dtype.

Parameters **series** (*pd.Index*) – One-dimensional ndarray with axis labels (including time series).

Return type Index

Returns Index with coerced data type

5.8.2.2.3 pandera.Index.validate

Index.**validate**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*)

Validate DataFrameSchema or SeriesSchema Index.

Check_obj pandas DataFrame of Series containing index to validate.

Parameters

- **head** (Optional[int]) – validate the first n rows. Rows overlapping with *tail* or *sample* are de-duplicated.
- **tail** (Optional[int]) – validate the last n rows. Rows overlapping with *head* or *sample* are de-duplicated.
- **sample** (Optional[int]) – validate a random sample of n rows. Rows overlapping with *head* or *tail* are de-duplicated.
- **random_state** (Optional[int]) – random seed for the sample argument.

Return type Union[DataFrame, Series]

Returns validated DataFrame or Series.

5.8.2.2.4 pandera.Index.__call__

Index.**__call__**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*)

Alias for validate method.

Return type Union[DataFrame, Series]

5.8.2.3 pandera.MultiIndex

class pandera.**MultiIndex**(*indexes*, *coerce=False*, *strict=False*, *name=None*)

Validate types and properties of a DataFrame MultiIndex.

Because *MultiIndex.__call__* converts the index to a dataframe via *to_frame()*, each index is treated as a series and it makes sense to inherit the *__call__* and *validate* methods from DataFrameSchema.

Create MultiIndex validator.

Parameters

- **indexes** (List[Index]) – list of Index validators for each level of the MultiIndex index.

- **coerce** (bool) – Whether or not to coerce the MultiIndex to the specified pandas_dtypes before validation
- **strict** (bool) – whether or not to accept columns in the MultiIndex that aren't defined in the indexes argument.
- **name** (Optional[str]) – name of schema component

Example

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>>
>>> schema = pa.DataFrameSchema(
...     columns={"column": pa.Column(pa.Int)},
...     index=pa.MultiIndex([
...         pa.Index(pa.String,
...                  pa.Check(lambda s: s.isin(["foo", "bar"])),
...                  name="index0"),
...         pa.Index(pa.Int, name="index1"),
...     ])
... )
>>>
>>> df = pd.DataFrame(
...     data={"column": [1, 2, 3]},
...     index=pd.MultiIndex.from_arrays(
...         [["foo", "bar", "foo"], [0, 1, 2]],
...         names=["index0", "index1"],
...     )
... )
>>>
>>> schema.validate(df)
               column
index0 index1
foo    0            1
bar    1            2
foo    2            3
```

See *here* for more usage details.

Attributes

coerce	Whether to coerce series to specified type.
dtype	A pandas style dtype dict where the keys are column names and values are pandas dtype for the column.

Methods

__init__	Create MultiIndex validator.
coerce_dtype	Coerce type of a pd.Series by type specified in pandas_dtype.
validate	Validate DataFrame or Series MultiIndex.
__call__	Alias for *DataFrameSchema.validate()* method.

5.8.2.3.1 pandera.MultiIndex.__init__

MultiIndex.**__init__**(*indexes*, *coerce=False*, *strict=False*, *name=None*)

Create MultiIndex validator.

Parameters

- **indexes** (List[Index]) – list of Index validators for each level of the MultiIndex index.
- **coerce** (bool) – Whether or not to coerce the MultiIndex to the specified pandas_dtypes before validation
- **strict** (bool) – whether or not to accept columns in the MultiIndex that aren't defined in the indexes argument.
- **name** (Optional[str]) – name of schema component

Example

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>>
>>> schema = pa.DataFrameSchema(
...     columns={"column": pa.Column(pa.Int)},
...     index=pa.MultiIndex([
...         pa.Index(pa.String,
...                  pa.Check(lambda s: s.isin(["foo", "bar"])),
...                  name="index0"),
...         pa.Index(pa.Int, name="index1"),
...     ])
... )
>>>
>>> df = pd.DataFrame(
...     data={"column": [1, 2, 3]},
...     index=pd.MultiIndex.from_arrays(
...         [["foo", "bar", "foo"], [0, 1, 2]],
...         names=["index0", "index1"],
...     )
... )
>>>
>>> schema.validate(df)
               column
index0 index1
foo    0            1
bar    1            2
foo    2            3
```

See *here* for more usage details.

Return type None

5.8.2.3.2 pandera.MultiIndex.coerce_dtype

MultiIndex.**coerce_dtype**(*multi_index*)

Coerce type of a pd.Series by type specified in pandas_dtype.

Parameters **multi_index** (MultiIndex) – multi-index to coerce.

Return type MultiIndex

Returns MultiIndex with coerced data type

5.8.2.3.3 pandera.MultiIndex.validate

MultiIndex.**validate**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*)

Validate DataFrame or Series MultiIndex.

Parameters

- **check_obj** (Union[DataFrame, Series]) – pandas DataFrame of Series to validate.
- **head** (Optional[int]) – validate the first n rows. Rows overlapping with *tail* or *sample* are de-duplicated.
- **tail** (Optional[int]) – validate the last n rows. Rows overlapping with *head* or *sample* are de-duplicated.
- **sample** (Optional[int]) – validate a random sample of n rows. Rows overlapping with *head* or *tail* are de-duplicated.
- **random_state** (Optional[int]) – random seed for the sample argument.
- **lazy** (bool) – if True, lazily evaluates dataframe against all validation checks and raises a SchemaErrorReport. Otherwise, raise SchemaError as soon as one occurs.

Return type Union[DataFrame, Series]

Returns validated DataFrame or Series.

5.8.2.3.4 pandera.MultiIndex.__call__

MultiIndex.**__call__**(*dataframe*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*)

Alias for *DataFrameSchema.validate()* method.

Parameters

- **dataframe** (*pd.DataFrame*) – the dataframe to be validated.
- **head** (*int*) – validate the first n rows. Rows overlapping with *tail* or *sample* are de-duplicated.
- **tail** (*int*) – validate the last n rows. Rows overlapping with *head* or *sample* are de-duplicated.

- **sample** (Optional[int]) – validate a random sample of n rows. Rows overlapping with *head* or *tail* are de-duplicated.
- **random_state** (Optional[int]) – random seed for the sample argument.
- **lazy** (bool) – if True, lazily evaluates dataframe against all validation checks and raises a SchemaErrorReport. Otherwise, raise SchemaError as soon as one occurs.

5.8.3 Checks

Check	Check a pandas Series or DataFrame for certain properties.
Hypothesis	Special type of *Check* that defines hypothesis tests on data.

5.8.3.1 pandera.Check

class pandera.**Check**(*check_fn*, *groups=None*, *groupby=None*, *ignore_na=True*, *element_wise=False*, *name=None*, *error=None*, *raise_warning=False*, *n_failure_cases=10*, ***check_kwargs*)

Check a pandas Series or DataFrame for certain properties.

Apply a validation function to each element, Series, or DataFrame.

Parameters

- **check_fn** (Callable) – A function to check pandas data structure. For Column or SeriesSchema checks, if element_wise is True, this function should have the signature: Callable[[pd.Series], Union[pd.Series, bool]], where the output series is a boolean vector.

  If element_wise is False, this function should have the signature: Callable[[Any], bool], where Any is an element in the column.

  For DataFrameSchema checks, if element_wise=True, fn should have the signature: Callable[[pd.DataFrame], Union[pd.DataFrame, pd.Series, bool]], where the output dataframe or series contains booleans.

  If element_wise is True, fn is applied to each row in the dataframe with the signature Callable[[pd.Series], bool] where the series input is a row in the dataframe.
- **groups** (Union[str, List[str], None]) – The dict input to the *fn* callable will be constrained to the groups specified by *groups*.
- **groupby** (Union[str, List[str], Callable, None]) – If a string or list of strings is provided, these columns are used to group the Column series. If a callable is passed, the expected signature is: Callable[[pd.DataFrame], pd.core.groupby.DataFrameGroupBy]

  The the case of Column checks, this function has access to the entire dataframe, but Column.name is selected from this DataFrameGroupby object so that a SeriesGroupBy object is passed into check_fn.

  Specifying the groupby argument changes the check_fn signature to:

  Callable[[Dict[Union[str, Tuple[str]], pd.Series]], Union[bool, pd.Series]] # noqa

where the input is a dictionary mapping keys to subsets of the column/dataframe.

- **ignore_na** (bool) – If True, drops null values on the checked series or dataframe before passing into the check_fn. For dataframes, drops rows with any null value. *New in version 0.4.0*
- **element_wise** (bool) – Whether or not to apply validator in an element-wise fashion. If bool, assumes that all checks should be applied to the column element-wise. If list, should be the same number of elements as checks.
- **name** (Optional[str]) – optional name for the check.
- **error** (Optional[str]) – custom error message if series fails validation check.
- **raise_warning** (bool) – if True, raise a UserWarning and do not throw exception instead of raising a SchemaError for a specific check. This option should be used carefully in cases where a failing check is informational and shouldn't stop execution of the program.
- **n_failure_cases** (Optional[int]) – report the top n failure cases. If None, then report all failure cases.
- **check_kwargs** – key-word arguments to pass into check_fn

Example

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>>
>>> # column checks are vectorized by default
>>> check_positive = pa.Check(lambda s: s > 0)
>>>
>>> # define an element-wise check
>>> check_even = pa.Check(lambda x: x % 2 == 0, element_wise=True)
>>>
>>> # specify assertions across categorical variables using `groupby`,
>>> # for example, make sure the mean measure for group "A" is always
>>> # larger than the mean measure for group "B"
>>> check_by_group = pa.Check(
...     lambda measures: measures["A"].mean() > measures["B"].mean(),
...     groupby=["group"],
... )
>>>
>>> # define a wide DataFrame-level check
>>> check_dataframe = pa.Check(
...     lambda df: df["measure_1"] > df["measure_2"])
>>>
>>> measure_checks = [check_positive, check_even, check_by_group]
>>>
>>> schema = pa.DataFrameSchema(
...     columns={
...         "measure_1": pa.Column(pa.Int, checks=measure_checks),
...         "measure_2": pa.Column(pa.Int, checks=measure_checks),
...         "group": pa.Column(pa.String),
...     },
...     checks=check_dataframe
... )
>>>
>>> df = pd.DataFrame({
...     "measure_1": [10, 12, 14, 16],
```

(continues on next page)

(continued from previous page)

```
...     "measure_2": [2, 4, 6, 8],
...     "group": ["B", "B", "A", "A"]
... })
>>>
>>> schema.validate(df)[["measure_1", "measure_2", "group"]]
   measure_1  measure_2 group
0         10          2     B
1         12          4     B
2         14          6     A
3         16          8     A
```

See *here* for more usage details.

Attributes

`statistics`	Get check statistics.

Methods

eq	Ensure all elements of a series equal a certain value.
equal_to	Ensure all elements of a series equal a certain value.
ge	Ensure all values are greater or equal a certain value.
greater_than	Ensure values of a series are strictly greater than a minimum value.
greater_than_or_equal_to	Ensure all values are greater or equal a certain value.
gt	Ensure values of a series are strictly greater than a minimum value.
in_range	Ensure all values of a series are within an interval.
isin	Ensure only allowed values occur within a series.
le	Ensure values are less than or equal to a maximum value.
less_than	Ensure values of a series are strictly below a maximum value.
less_than_or_equal_to	Ensure values are less than or equal to a maximum value.
lt	Ensure values of a series are strictly below a maximum value.
ne	Ensure no elements of a series equals a certain value.
not_equal_to	Ensure no elements of a series equals a certain value.
notin	Ensure some defined values don't occur within a series.
str_contains	Ensure that a pattern can be found within each row.
str_endswith	Ensure that all values end with a certain string.
str_length	Ensure that the length of strings is within a specified range.
str_matches	Ensure that string values match a regular expression.
str_startswith	Ensure that all values start with a certain string.
__call__	Validate pandas DataFrame or Series.

5.8.3.1.1 pandera.Check.eq

classmethod Check.**eq**(*cls*, *value*, ***kwargs*)
Ensure all elements of a series equal a certain value.

New in version 0.4.5 Alias: `eq`

Parameters

- **value** – All elements of a given `pandas.Series` must have this value
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type `Check`

Returns *Check* object

5.8.3.1.2 pandera.Check.equal_to

classmethod Check.**equal_to**(*cls*, *value*, ***kwargs*)
Ensure all elements of a series equal a certain value.

New in version 0.4.5 Alias: `eq`

Parameters

- **value** – All elements of a given `pandas.Series` must have this value
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type `Check`

Returns *Check* object

5.8.3.1.3 pandera.Check.ge

classmethod Check.**ge**(*cls*, *min_value*, ***kwargs*)
Ensure all values are greater or equal a certain value.

New in version 0.4.5 Alias: `ge`

Parameters

- **min_value** – Allowed minimum value for values of a series. Must be a type comparable to the dtype of the `pandas.Series` to be validated.
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type `Check`

Returns *Check* object

5.8.3.1.4 pandera.Check.greater_than

classmethod Check.**greater_than**(*cls*, *min_value*, ***kwargs*)

Ensure values of a series are strictly greater than a minimum value.

New in version 0.4.5 Alias: `gt`

Parameters

- **min_value** – Lower bound to be exceeded. Must be a type comparable to the dtype of the `pandas.Series` to be validated (e.g. a numerical type for float or int and a datetime for datetime).
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type Check

Returns *Check* object

5.8.3.1.5 pandera.Check.greater_than_or_equal_to

classmethod Check.**greater_than_or_equal_to**(*cls*, *min_value*, ***kwargs*)

Ensure all values are greater or equal a certain value.

New in version 0.4.5 Alias: `ge`

Parameters

- **min_value** – Allowed minimum value for values of a series. Must be a type comparable to the dtype of the `pandas.Series` to be validated.
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type Check

Returns *Check* object

5.8.3.1.6 pandera.Check.gt

classmethod Check.**gt**(*cls*, *min_value*, ***kwargs*)

Ensure values of a series are strictly greater than a minimum value.

New in version 0.4.5 Alias: `gt`

Parameters

- **min_value** – Lower bound to be exceeded. Must be a type comparable to the dtype of the `pandas.Series` to be validated (e.g. a numerical type for float or int and a datetime for datetime).
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type Check

Returns *Check* object

5.8.3.1.7 pandera.Check.in_range

classmethod Check.**in_range**(*cls*, *min_value*, *max_value*, *include_min=True*, *include_max=True*, ***kwargs*)

Ensure all values of a series are within an interval.

Parameters

- **min_value** – Left / lower endpoint of the interval.
- **max_value** – Right / upper endpoint of the interval. Must not be smaller than min_value.
- **include_min** – Defines whether min_value is also an allowed value (the default) or whether all values must be strictly greater than min_value.
- **include_max** – Defines whether min_value is also an allowed value (the default) or whether all values must be strictly smaller than max_value.
- **kwargs** – key-word arguments passed into the *Check* initializer.

Both endpoints must be a type comparable to the dtype of the pandas.Series to be validated.

Return type Check

Returns *Check* object

5.8.3.1.8 pandera.Check.isin

classmethod Check.**isin**(*cls*, *allowed_values*, ***kwargs*)

Ensure only allowed values occur within a series.

Parameters

- **allowed_values** (Iterable) – The set of allowed values. May be any iterable.
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type Check

Returns *Check* object

Note: It is checked whether all elements of a pandas.Series are part of the set of elements of allowed values. If allowed values is a string, the set of elements consists of all distinct characters of the string. Thus only single characters which occur in allowed_values at least once can meet this condition. If you want to check for substrings use Check.str_is_substring().

5.8.3.1.9 pandera.Check.le

classmethod Check.**le**(*cls*, *max_value*, ***kwargs*)

Ensure values are less than or equal to a maximum value.

New in version 0.4.5 Alias: le

Parameters

- **max_value** – Upper bound not to be exceeded. Must be a type comparable to the dtype of the pandas.Series to be validated.
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type `Check`

Returns *Check* object

5.8.3.1.10 pandera.Check.less_than

classmethod `Check.`**less_than**(*cls*, *max_value*, ***kwargs*)

Ensure values of a series are strictly below a maximum value.

New in version 0.4.5 Alias: `lt`

Parameters

- **max_value** – All elements of a series must be strictly smaller than this. Must be a type comparable to the dtype of the `pandas.Series` to be validated.
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type `Check`

Returns *Check* object

5.8.3.1.11 pandera.Check.less_than_or_equal_to

classmethod `Check.`**less_than_or_equal_to**(*cls*, *max_value*, ***kwargs*)

Ensure values are less than or equal to a maximum value.

New in version 0.4.5 Alias: `le`

Parameters

- **max_value** – Upper bound not to be exceeded. Must be a type comparable to the dtype of the `pandas.Series` to be validated.
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type `Check`

Returns *Check* object

5.8.3.1.12 pandera.Check.lt

classmethod `Check.`**lt**(*cls*, *max_value*, ***kwargs*)

Ensure values of a series are strictly below a maximum value.

New in version 0.4.5 Alias: `lt`

Parameters

- **max_value** – All elements of a series must be strictly smaller than this. Must be a type comparable to the dtype of the `pandas.Series` to be validated.
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type `Check`

Returns *Check* object

5.8.3.1.13 pandera.Check.ne

classmethod Check.**ne**(*cls*, *value*, ***kwargs*)
Ensure no elements of a series equals a certain value.

New in version 0.4.5 Alias: ne

Parameters

- **value** – This value must not occur in the checked pandas.Series.
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type Check

Returns *Check* object

5.8.3.1.14 pandera.Check.not_equal_to

classmethod Check.**not_equal_to**(*cls*, *value*, ***kwargs*)
Ensure no elements of a series equals a certain value.

New in version 0.4.5 Alias: ne

Parameters

- **value** – This value must not occur in the checked pandas.Series.
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type Check

Returns *Check* object

5.8.3.1.15 pandera.Check.notin

classmethod Check.**notin**(*cls*, *forbidden_values*, ***kwargs*)
Ensure some defined values don't occur within a series.

Parameters

- **forbidden_values** (Iterable) – The set of values which should not occur. May be any iterable.
- **raise_warning** – if True, check raises UserWarning instead of SchemaError on validation.

Return type Check

Returns *Check* object

Note: Like *Check.isin()* this check operates on single characters if it is applied on strings. A string as paraforbidden_valuesmeter forbidden_values is understood as set of prohibited characters. Any string of length > 1 can't be in it by design.

5.8.3.1.16 pandera.Check.str_contains

classmethod Check.**str_contains**(*cls*, *pattern*, ***kwargs*)
Ensure that a pattern can be found within each row.

Parameters

- **pattern** (str) – Regular expression pattern to use for searching
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type Check

Returns *Check* object

The behaviour is as of pandas.Series.str.contains().

5.8.3.1.17 pandera.Check.str_endswith

classmethod Check.**str_endswith**(*cls*, *string*, ***kwargs*)
Ensure that all values end with a certain string.

Parameters

- **string** (str) – String all values should end with
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type Check

Returns *Check* object

5.8.3.1.18 pandera.Check.str_length

classmethod Check.**str_length**(*cls*, *min_value=None*, *max_value=None*, ***kwargs*)
Ensure that the length of strings is within a specified range.

Parameters

- **min_value** (Optional[int]) – Minimum length of strings (default: no minimum)
- **max_value** (Optional[int]) – Maximum length of strings (default: no maximum)
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type Check

Returns *Check* object

5.8.3.1.19 pandera.Check.str_matches

classmethod Check.**str_matches**(*cls*, *pattern*, ***kwargs*)
Ensure that string values match a regular expression.

Parameters

- **pattern** (str) – Regular expression pattern to use for matching
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type Check

Returns *Check* object

The behaviour is as of `pandas.Series.str.match()`.

5.8.3.1.20 pandera.Check.str_startswith

classmethod Check.**str_startswith**(*cls*, *string*, ***kwargs*)

Ensure that all values start with a certain string.

Parameters

- **string** (str) – String all values should start with
- **kwargs** – key-word arguments passed into the *Check* initializer.

Return type Check

Returns *Check* object

5.8.3.1.21 pandera.Check.__call__

Check.**__call__**(*df_or_series*, *column=None*)

Validate pandas DataFrame or Series.

Parameters

- **df_or_series** (Union[DataFrame, Series]) – pandas DataFrame of Series to validate.
- **column** (Optional[str]) – for dataframe checks, apply the check function to this column.

Return type CheckResult

Returns

CheckResult tuple containing:

`check_output`: boolean scalar, `Series` or `DataFrame` indicating which elements passed the check.

`check_passed`: boolean scalar that indicating whether the check passed overall.

`checked_object`: the checked object itself. Depending on the options provided to the `Check`, this will be a pandas Series, DataFrame, or if the `groupby` option is specified, a `Dict[str, Series]` or `Dict[str, DataFrame]` where the keys are distinct groups.

`failure_cases`: subset of the check_object that failed.

5.8.3.2 pandera.Hypothesis

class pandera.**Hypothesis**(*test*, *samples=None*, *groupby=None*, *relationship='equal'*, *test_kwargs=None*, *relationship_kwargs=None*, *name=None*, *error=None*, *raise_warning=False*)

Special type of `Check` that defines hypothesis tests on data.

Perform a hypothesis test on a Series or DataFrame.

Parameters

- **test** (`Callable`) – The hypothesis test function. It should take one or more arrays as positional arguments and return a test statistic and a p-value. The arrays passed into the test function are determined by the `samples` argument.
- **samples** (`Union[str, List[str], None]`) – for *Column* or *SeriesSchema* hypotheses, this refers to the group keys in the *groupby* column(s) used to group the *Series* into a dict of *Series*. The *samples* column(s) are passed into the *test* function as positional arguments.

  For *DataFrame*-level hypotheses, *samples* refers to a column or multiple columns to pass into the *test* function. The *samples* column(s) are passed into the *test* function as positional arguments.
- **groupby** (`Union[str, List[str], Callable, None]`) – If a string or list of strings is provided, then these columns are used to group the Column Series by *groupby*. If a callable is passed, the expected signature is DataFrame -> DataFrameGroupby. The function has access to the entire dataframe, but the Column.name is selected from this DataFrameGroupby object so that a SeriesGroupBy object is passed into the *hypothesis_check* function.

  Specifying this argument changes the *fn* signature to: dict[str|tuple[str], Series] -> bool|pd.Series[bool]

  Where specific groups can be obtained from the input dict.
- **relationship** (`Union[str, Callable]`) – Represents what relationship conditions are imposed on the hypothesis test. A function or lambda function can be supplied.

  Available built-in relationships are: "greater_than", "less_than", "not_equal" or "equal", where "equal" is the null hypothesis.

  If callable, the input function signature should have the signature `(stat: float, pvalue: float, **kwargs)` where *stat* is the hypothesis test statistic, *pvalue* assesses statistical significance, and ***kwargs* are other arguments supplied via the ***relationship_kwargs* argument.

  Default is "equal" for the null hypothesis.
- **test_kwargs** (*dict*) – Keyword arguments to be supplied to the test.
- **relationship_kwargs** (*dict*) – Keyword arguments to be supplied to the relationship function. e.g. *alpha* could be used to specify a threshold in a t-test.
- **name** (`Optional[str]`) – optional name of hypothesis test
- **error** (`Optional[str]`) – error message to show
- **raise_warning** (`bool`) – if True, raise a UserWarning and do not throw exception instead of raising a SchemaError for a specific check. This option should be used carefully in cases where a failing check is informational and shouldn't stop execution of the program.

Examples

Define a two-sample hypothesis test using scipy.

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>> from scipy import stats
>>>
>>> schema = pa.DataFrameSchema({
...     "height_in_feet": pa.Column(pa.Float, [
...         pa.Hypothesis(
...             test=stats.ttest_ind,
...             samples=["A", "B"],
...             groupby="group",
...             # assert that the mean height of group "A" is greater
...             # than that of group "B"
...             relationship=lambda stat, pvalue, alpha=0.1: (
...                 stat > 0 and pvalue / 2 < alpha
...             ),
...             # set alpha criterion to 5%
...             relationship_kwargs={"alpha": 0.05}
...         )
...     ]),
...     "group": pa.Column(pa.String),
... })
>>> df = (
...     pd.DataFrame({
...         "height_in_feet": [8.1, 7, 5.2, 5.1, 4],
...         "group": ["A", "A", "B", "B", "B"]
...     })
... )
>>> schema.validate(df)[["height_in_feet", "group"]]
   height_in_feet group
0             8.1     A
1             7.0     A
2             5.2     B
3             5.1     B
4             4.0     B
```

See *here* for more usage details.

Attributes

RELATIONSHIPS	Relationships available for built-in hypothesis tests.
is_one_sample_test	Return True if hypothesis is a one-sample test.
statistics	Get check statistics.

Methods

__init__	Perform a hypothesis test on a Series or DataFrame.
one_sample_ttest	Calculate a t-test for the mean of one sample.
two_sample_ttest	Calculate a t-test for the means of two samples.
__call__	Validate pandas DataFrame or Series.

5.8.3.2.1 pandera.Hypothesis.__init__

Hypothesis.**__init__**(*test*, *samples=None*, *groupby=None*, *relationship='equal'*, *test_kwargs=None*, *relationship_kwargs=None*, *name=None*, *error=None*, *raise_warning=False*)

Perform a hypothesis test on a Series or DataFrame.

Parameters

- **test** (Callable) – The hypothesis test function. It should take one or more arrays as positional arguments and return a test statistic and a p-value. The arrays passed into the test function are determined by the samples argument.
- **samples** (Union[str, List[str], None]) – for *Column* or *SeriesSchema* hypotheses, this refers to the group keys in the *groupby* column(s) used to group the *Series* into a dict of *Series*. The *samples* column(s) are passed into the *test* function as positional arguments.

  For *DataFrame*-level hypotheses, *samples* refers to a column or multiple columns to pass into the *test* function. The *samples* column(s) are passed into the *test* function as positional arguments.
- **groupby** (Union[str, List[str], Callable, None]) – If a string or list of strings is provided, then these columns are used to group the Column Series by *groupby*. If a callable is passed, the expected signature is DataFrame -> DataFrameGroupby. The function has access to the entire dataframe, but the Column.name is selected from this DataFrameGroupby object so that a SeriesGroupBy object is passed into the *hypothesis_check* function.

  Specifying this argument changes the *fn* signature to: dict[str|tuple[str], Series] -> bool|pd.Series[bool]

  Where specific groups can be obtained from the input dict.
- **relationship** (Union[str, Callable]) – Represents what relationship conditions are imposed on the hypothesis test. A function or lambda function can be supplied.

  Available built-in relationships are: “greater_than”, “less_than”, “not_equal” or “equal”, where “equal” is the null hypothesis.

  If callable, the input function signature should have the signature (stat: float, pvalue: float, **kwargs) where *stat* is the hypothesis test statistic, *pvalue* assesses statistical significance, and ***kwargs* are other arguments supplied via the ***relationship_kwargs* argument.

  Default is “equal” for the null hypothesis.
- **test_kwargs** (*dict*) – Keyword arguments to be supplied to the test.
- **relationship_kwargs** (*dict*) – Keyword arguments to be supplied to the relationship function. e.g. *alpha* could be used to specify a threshold in a t-test.
- **name** (Optional[str]) – optional name of hypothesis test

- **error** (Optional[str]) – error message to show
- **raise_warning** (bool) – if True, raise a UserWarning and do not throw exception instead of raising a SchemaError for a specific check. This option should be used carefully in cases where a failing check is informational and shouldn't stop execution of the program.

Examples

Define a two-sample hypothesis test using scipy.

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>> from scipy import stats
>>>
>>> schema = pa.DataFrameSchema({
...     "height_in_feet": pa.Column(pa.Float, [
...         pa.Hypothesis(
...             test=stats.ttest_ind,
...             samples=["A", "B"],
...             groupby="group",
...             # assert that the mean height of group "A" is greater
...             # than that of group "B"
...             relationship=lambda stat, pvalue, alpha=0.1: (
...                 stat > 0 and pvalue / 2 < alpha
...             ),
...             # set alpha criterion to 5%
...             relationship_kwargs={"alpha": 0.05}
...         )
...     ]),
...     "group": pa.Column(pa.String),
... })
>>> df = (
...     pd.DataFrame({
...         "height_in_feet": [8.1, 7, 5.2, 5.1, 4],
...         "group": ["A", "A", "B", "B", "B"]
...     })
... )
>>> schema.validate(df)[["height_in_feet", "group"]]
   height_in_feet group
0             8.1     A
1             7.0     A
2             5.2     B
3             5.1     B
4             4.0     B
```

See *here* for more usage details.

Return type None

5.8.3.2.2 pandera.Hypothesis.one_sample_ttest

classmethod Hypothesis.**one_sample_ttest**(*popmean*, *sample=None*, *groupby=None*, *relationship='equal'*, *alpha=0.01*, *raise_warning=False*)

Calculate a t-test for the mean of one sample.

Parameters

- **sample** (Optional[str]) – The sample group to test. For *Column* and *SeriesSchema* hypotheses, this refers to the *groupby* level that is used to subset the *Column* being checked. For *DataFrameSchema* hypotheses, refers to column in the *DataFrame*.
- **groupby** (Union[str, List[str], Callable, None]) – If a string or list of strings is provided, then these columns are used to group the Column Series by *groupby*. If a callable is passed, the expected signature is DataFrame -> DataFrameGroupby. The function has access to the entire dataframe, but the Column.name is selected from this DataFrameGroupby object so that a SeriesGroupBy object is passed into *fn*.

  Specifying this argument changes the *fn* signature to: dict[str|tuple[str], Series] -> bool|pd.Series[bool]

  Where specific groups can be obtained from the input dict.
- **popmean** (float) – population mean to compare *sample* to.
- **relationship** (str) – Represents what relationship conditions are imposed on the hypothesis test. Available relationships are: "greater_than", "less_than", "not_equal" and "equal". For example, *group1 greater_than group2* specifies an alternative hypothesis that the mean of group1 is greater than group 2 relative to a null hypothesis that they are equal.
- **alpha** (float) – (Default value = 0.01) The significance level; the probability of rejecting the null hypothesis when it is true. For example, a significance level of 0.01 indicates a 1% risk of concluding that a difference exists when there is no actual difference.
- **raise_warning** – if True, check raises UserWarning instead of SchemaError on validation.

Example

If you want to compare one sample with a pre-defined mean:

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>>
>>> schema = pa.DataFrameSchema({
...     "height_in_feet": pa.Column(
...         pa.Float, [
...             pa.Hypothesis.one_sample_ttest(
...                 popmean=5,
...                 relationship="greater_than",
...                 alpha=0.1),
...     ]),
... })
>>> df = (
...     pd.DataFrame({
...         "height_in_feet": [8.1, 7, 6.5, 6.7, 5.1],
...     })
... )
```

(continues on next page)

(continued from previous page)

```
>>> schema.validate(df)
   height_in_feet
0             8.1
1             7.0
2             6.5
3             6.7
4             5.1
```

5.8.3.2.3 pandera.Hypothesis.two_sample_ttest

classmethod Hypothesis.**two_sample_ttest**(*sample1*, *sample2*, *groupby=None*, *relationship='equal'*, *alpha=0.01*, *equal_var=True*, *nan_policy='propagate'*, *raise_warning=False*)

Calculate a t-test for the means of two samples.

Perform a two-sided test for the null hypothesis that 2 independent samples have identical average (expected) values. This test assumes that the populations have identical variances by default.

Parameters

- **sample1** (str) – The first sample group to test. For *Column* and *SeriesSchema* hypotheses, refers to the level in the *groupby* column. For *DataFrameSchema* hypotheses, refers to column in the *DataFrame*.
- **sample2** (str) – The second sample group to test. For *Column* and *SeriesSchema* hypotheses, refers to the level in the *groupby* column. For *DataFrameSchema* hypotheses, refers to column in the *DataFrame*.
- **groupby** (Union[str, List[str], Callable, None]) – If a string or list of strings is provided, then these columns are used to group the Column Series by *groupby*. If a callable is passed, the expected signature is DataFrame -> DataFrameGroupby. The function has access to the entire dataframe, but the Column.name is selected from this DataFrameGroupby object so that a SeriesGroupBy object is passed into *fn*.

  Specifying this argument changes the *fn* signature to: dict[str|tuple[str], Series] -> bool|pd.Series[bool]

  Where specific groups can be obtained from the input dict.
- **relationship** (str) – Represents what relationship conditions are imposed on the hypothesis test. Available relationships are: "greater_than", "less_than", "not_equal", and "equal". For example, *group1 greater_than group2* specifies an alternative hypothesis that the mean of group1 is greater than group 2 relative to a null hypothesis that they are equal.
- **alpha** – (Default value = 0.01) The significance level; the probability of rejecting the null hypothesis when it is true. For example, a significance level of 0.01 indicates a 1% risk of concluding that a difference exists when there is no actual difference.
- **equal_var** – (Default value = True) If True (default), perform a standard independent 2 sample test that assumes equal population variances. If False, perform Welch's t-test, which does not assume equal population variance
- **nan_policy** – Defines how to handle when input returns nan, one of {'propagate', 'raise', 'omit'}, (Default value = 'propagate'). For more details see: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html

- **raise_warning** – if True, check raises UserWarning instead of SchemaError on validation.

Example

The the built-in class method to do a two-sample t-test.

```
>>> import pandera as pa
>>>
>>>
>>> schema = pa.DataFrameSchema({
...     "height_in_feet": pa.Column(
...         pa.Float, [
...             pa.Hypothesis.two_sample_ttest(
...                 sample1="A",
...                 sample2="B",
...                 groupby="group",
...                 relationship="greater_than",
...                 alpha=0.05,
...                 equal_var=True),
...     ]),
...     "group": pa.Column(pa.String)
... })
>>> df = (
...     pd.DataFrame({
...         "height_in_feet": [8.1, 7, 5.2, 5.1, 4],
...         "group": ["A", "A", "B", "B", "B"]
...     })
... )
>>> schema.validate(df)[["height_in_feet", "group"]]
   height_in_feet group
0             8.1     A
1             7.0     A
2             5.2     B
3             5.1     B
4             4.0     B
```

5.8.3.2.4 pandera.Hypothesis.__call__

Hypothesis.**__call__**(*df_or_series*, *column=None*)

Validate pandas DataFrame or Series.

Parameters

- **df_or_series** (Union[DataFrame, Series]) – pandas DataFrame of Series to validate.
- **column** (Optional[str]) – for dataframe checks, apply the check function to this column.

Return type CheckResult

Returns

CheckResult tuple containing:

check_output: boolean scalar, Series or DataFrame indicating which elements passed the check.

check_passed: boolean scalar that indicating whether the check passed overall.

`checked_object`: the checked object itself. Depending on the options provided to the `Check`, this will be a pandas Series, DataFrame, or if the `groupby` option is specified, a `Dict[str, Series]` or `Dict[str, DataFrame]` where the keys are distinct groups.

`failure_cases`: subset of the check_object that failed.

5.8.4 Pandas Data Types

PandasDtype	Enumerate all valid pandas data types.

5.8.4.1 pandera.PandasDtype

class pandera.**PandasDtype**(*value*)

Bases: `enum.Enum`

Enumerate all valid pandas data types.

`pandera` follows the numpy data types subscribed to by `pandas` and by default supports using the numpy data type string aliases to validate DataFrame or Series dtypes.

This class simply enumerates the valid numpy dtypes for pandas arrays. For convenience `PandasDtype` enums can all be accessed in the top-level `pandera` name space via the same enum name.

Examples

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>>
>>> pa.SeriesSchema(pa.Int).validate(pd.Series([1, 2, 3]))
0    1
1    2
2    3
dtype: int64
>>> pa.SeriesSchema(pa.Float).validate(pd.Series([1.1, 2.3, 3.4]))
0    1.1
1    2.3
2    3.4
dtype: float64
>>> pa.SeriesSchema(pa.String).validate(pd.Series(["a", "b", "c"]))
    0    a
1    b
2    c
dtype: object
```

Alternatively, you can use built-in python scalar types for integers, floats, booleans, and strings:

```
>>> pa.SeriesSchema(int).validate(pd.Series([1, 2, 3]))
0    1
1    2
2    3
dtype: int64
```

You can also use the pandas string aliases in the schema definition:

```
>>> pa.SeriesSchema("int").validate(pd.Series([1, 2, 3]))
0    1
1    2
2    3
dtype: int64
```

Note: `pandera` also offers limited support for pandas extension types, however since the release of pandas 1.0.0 there are backwards incompatible extension types like the `Integer` array. The extension types, e.g. `pd.IntDtype64()` and their string alias should work when supplied to the `pandas_dtype` argument, unless otherwise specified below, but this functionality is only tested for pandas >= 1.0.0. Extension types in earlier versions are not guaranteed to work as the `pandas_dtype` argument in schemas or schema components.

Attributes

`Bool`	`"bool"` numpy dtype
`Category`	pandas `"categorical"` datatype
`DateTime`	`"datetime64[ns]"` numpy dtype
`Float`	`"float"` numpy dtype
`Float16`	`"float16"` numpy dtype
`Float32`	`"float32"` numpy dtype
`Float64`	`"float64"` numpy dtype
`INT16`	`"Int16"` pandas dtype: pandas 0.24.0+
`INT32`	`"Int32"` pandas dtype: pandas 0.24.0+
`INT64`	`"Int64"` pandas dtype: pandas 0.24.0+
`INT8`	`"Int8"` pandas dtype:: pandas 0.24.0+
`Int`	`"int"` numpy dtype
`Int16`	`"int16"` numpy dtype
`Int32`	`"int32"` numpy dtype
`Int64`	`"int64"` numpy dtype
`Int8`	`"int8"` numpy dtype
`Object`	`"object"` numpy dtype
`String`	The string datatype doesn't map to the first-class pandas datatype and is representated as a numpy `"object"` array.
`Timedelta`	`"timedelta64[ns]"` numpy dtype
`UINT16`	`"UInt16"` pandas dtype: pandas 0.24.0+
`UINT32`	`"UInt32"` pandas dtype: pandas 0.24.0+
`UINT64`	`"UInt64"` pandas dtype: pandas 0.24.0+
`UINT8`	`"UInt8"` pandas dtype:: pandas 0.24.0+
`UInt16`	`"uint16"` numpy dtype
`UInt32`	`"uint32"` numpy dtype
`UInt64`	`"uint64"` numpy dtype
`UInt8`	`"uint8"` numpy dtype

str_alias
Get datatype string alias.

classmethod from_str_alias(*str_alias*)
Get PandasDtype from string alias.

Param pandas dtype string alias from https://pandas.pydata.org/pandas-docs/stable/getting_started/basics.html#basics-dtypes

Return type PandasDtype

Returns pandas dtype

classmethod from_pandas_api_type(*pandas_api_type*)

Get PandasDtype enum from pandas api type.

Parameters **pandas_api_type** (str) – string output from https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.api.types.infer_dtype.html

Return type PandasDtype

Returns pandas dtype

5.8.5 Decorators

check_input	Validate function argument when function is called.
check_output	Validate function output.

5.8.5.1 pandera.check_input

pandera.**check_input**(*schema*, *obj_getter=None*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*)

Validate function argument when function is called.

This is a decorator function that validates the schema of a dataframe argument in a function. Note that if a transformer is specified by the schema, the decorator will return the transformed dataframe, which will be passed into the decorated function.

Parameters

- **schema** (Union[DataFrameSchema, SeriesSchema]) – dataframe/series schema object
- **obj_getter** (Union[str, int, None]) – (Default value = None) if int, obj_getter refers to the the index of the pandas dataframe/series to be validated in the args part of the function signature. If str, obj_getter refers to the argument name of the pandas dataframe/series in the function signature. This works even if the series/dataframe is passed in as a positional argument when the function is called. If None, assumes that the dataframe/series is the first argument of the decorated function
- **head** (Optional[int]) – validate the first n rows. Rows overlapping with *tail* or *sample* are de-duplicated.
- **tail** (Optional[int]) – validate the last n rows. Rows overlapping with *head* or *sample* are de-duplicated.
- **sample** (Optional[int]) – validate a random sample of n rows. Rows overlapping with *head* or *tail* are de-duplicated.
- **random_state** (Optional[int]) – random seed for the sample argument.
- **lazy** (bool) – if True, lazily evaluates dataframe against all validation checks and raises a SchemaErrorReport. Otherwise, raise SchemaError as soon as one occurs.

Return type Callable

Returns wrapped function

Example

Check the input of a decorated function.

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>>
>>> schema = pa.DataFrameSchema({"column": pa.Column(pa.Int)})
>>>
>>> @pa.check_input(schema)
... def transform_data(df: pd.DataFrame) -> pd.DataFrame:
...     df["doubled_column"] = df["column"] * 2
...     return df
>>>
>>> df = pd.DataFrame({
...     "column": range(5),
... })
>>>
>>> transform_data(df)
   column  doubled_column
0       0               0
1       1               2
2       2               4
3       3               6
4       4               8
```

See *here* for more usage details.

5.8.5.2 pandera.check_output

pandera.**check_output**(*schema*, *obj_getter=None*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*)

Validate function output.

Similar to input validator, but validates the output of the decorated function. Note that the *transformer* function supplied to the DataFrameSchema will not have an effect in the check_output schema validator.

Parameters

- **schema** (Union[DataFrameSchema, SeriesSchema]) – dataframe/series schema object
- **obj_getter** (Union[int, str, Callable, None]) – (Default value = None) if int, assumes that the output of the decorated function is a list-like object, where obj_getter is the index of the pandas data dataframe/series to be validated. If str, expects that the output is a dict-like object, and obj_getter is the key pointing to the dataframe/series to be validated. If a callable is supplied, it expects the output of decorated function and should return the dataframe/series to be validated.
- **head** (Optional[int]) – validate the first n rows. Rows overlapping with *tail* or *sample* are de-duplicated.
- **tail** (Optional[int]) – validate the last n rows. Rows overlapping with *head* or *sample* are de-duplicated.
- **sample** (Optional[int]) – validate a random sample of n rows. Rows overlapping with *head* or *tail* are de-duplicated.

- **random_state** (Optional[int]) – random seed for the sample argument.
- **lazy** (bool) – if True, lazily evaluates dataframe against all validation checks and raises a SchemaErrorReport. Otherwise, raise SchemaError as soon as one occurs.

Return type Callable

Returns wrapped function

Example

Check the output a decorated function.

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>>
>>> schema = pa.DataFrameSchema(
...     columns={"doubled_column": pa.Column(pa.Int)},
...     checks=pa.Check(
...         lambda df: df["doubled_column"] == df["column"] * 2
...     )
... )
>>>
>>> @pa.check_output(schema)
... def transform_data(df: pd.DataFrame) -> pd.DataFrame:
...     df["doubled_column"] = df["column"] * 2
...     return df
>>>
>>> df = pd.DataFrame({"column": range(5)})
>>>
>>> transform_data(df)
   column  doubled_column
0       0               0
1       1               2
2       2               4
3       3               6
4       4               8
```

See *here* for more usage details.

5.8.6 Schema Inference

infer_schema	Infer schema for pandas DataFrame or Series object.

5.8.6.1 pandera.infer_schema

pandera.**infer_schema**(*pandas_obj*)

Infer schema for pandas DataFrame or Series object.

Parameters **pandas_obj** (Union[DataFrame, Series]) – DataFrame or Series object to infer.

Return type Union[DataFrameSchema, SeriesSchema]

Returns DataFrameSchema or SeriesSchema

Raises TypeError if pandas_obj is not expected type.

5.8.7 IO Utils

io.from_yaml	Create *DataFrameSchema* from yaml file.
io.to_yaml	Write *DataFrameSchema* to yaml file.
io.to_script	Write *DataFrameSchema* to a python script.

5.8.7.1 pandera.io.from_yaml

pandera.io.**from_yaml**(*yaml_schema*)

Create DataFrameSchema from yaml file.

Parameters **yaml_schema** – str or Path to yaml schema, or serialized yaml string.

Returns dataframe schema.

5.8.7.2 pandera.io.to_yaml

pandera.io.**to_yaml**(*dataframe_schema*, *stream=None*)

Write DataFrameSchema to yaml file.

Parameters

- **dataframe_schema** – schema to write to file or dump to string.
- **stream** – file stream to write to. If None, dumps to string.

Returns yaml string if stream is None, otherwise returns None.

5.8.7.3 pandera.io.to_script

pandera.io.**to_script**(*dataframe_schema*, *path_or_buf=None*)

Write DataFrameSchema to a python script.

Parameters

- **dataframe_schema** – schema to write to file or dump to string.
- **path_or_buf** – filepath or buf stream to write to. If None, outputs string representation of the script.

Returns yaml string if stream is None, otherwise returns None.

5.8.8 Errors

errors.SchemaError	Raised when object does not pass schema validation constraints.
errors.SchemaErrors	Raised when multiple schema are lazily collected into one error.
errors.SchemaInitError	Raised when schema initialization fails.
errors.SchemaDefinitionError	Raised when schema definition is invalid on object validation.

5.8.8.1 pandera.errors.SchemaError

class pandera.errors.**SchemaError**(*schema*, *data*, *message*, *failure_cases=None*, *check=None*, *check_index=None*)

Raised when object does not pass schema validation constraints.

5.8.8.2 pandera.errors.SchemaErrors

class pandera.errors.**SchemaErrors**(*schema_error_dicts*, *data*)

Raised when multiple schema are lazily collected into one error.

5.8.8.3 pandera.errors.SchemaInitError

class pandera.errors.**SchemaInitError**

Raised when schema initialization fails.

5.8.8.4 pandera.errors.SchemaDefinitionError

class pandera.errors.**SchemaDefinitionError**

Raised when schema definition is invalid on object validation.

CHAPTER

SIX

INDICES AND TABLES

- genindex

INDEX

Symbols

A

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V