pandera

Niels Bantilan, Nigel Markey, Jean-Francois Zinque

Jul 13, 2021

INTRODUCTION

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*.
4. Define schema models with the *class-based API* with pydantic-style syntax and validate dataframes using the typing syntax.
5. *Synthesize data* from schema objects for property-based testing with pandas data structures.

CHAPTER

ONE

INSTALL

Install with *pip*:

```
pip install pandera
```

Installing optional functionality:

```
pip install pandera[hypotheses]  # hypothesis checks
pip install pandera[io]          # yaml/script schema io utilities
pip install pandera[strategies]  # data synthesis strategies
pip install pandera[all]         # all packages
```

Or conda:

```
conda install -c conda-forge pandera-core  # core library functionality
conda install -c conda-forge pandera       # pandera with all extensions
```

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.le(10)),
    "column2": pa.Column(float, checks=pa.Check.lt(-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
```

You can pass the built-in python types that are supported by pandas, or strings representing the legal pandas datatypes, or pandera's `PandasDtype` enum:

```
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("str"),

    # pandera PandasDtype enum
    "int_column3": pa.Column(pa.Int),
    "float_column3": pa.Column(pa.Float),
    "str_column3": pa.Column(pa.String),
})
```

For more details on data types, see *PandasDtype*

CHAPTER

THREE

SCHEMA MODEL

pandera also provides an alternative API for expressing schemas inspired by dataclasses and pydantic. The equivalent *SchemaModel* for the above DataFrameSchema would be:

```
from pandera.typing import Series

class Schema(pa.SchemaModel):

    column1: Series[int] = pa.Field(le=10)
    column2: Series[float] = pa.Field(lt=-1.2)
    column3: Series[str] = pa.Field(str_startswith="value_")

    @pa.check("column3")
    def column_3_check(cls, series: Series[str]) -> Series[bool]:
        """Check that column3 values have two elements after being split with '_'"""
        return series.str.split("_", expand=True).shape[1] == 2

Schema.validate(df)
```

CHAPTER

FOUR

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

FIVE

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

SIX

ISSUES

Submit issues, feature requests or bugfixes on github.

6.1 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,
)
```

6.1.1 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.

6.1.1.1 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
```

6.1.1.2 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):
...
pandera.errors.SchemaError: Error while coercing 'column1' to type int64: Cannot␣
↪convert non-finite values (NA or inf) 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)`.

6.1.1.3 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
```

6.1.1.4 Ordered Columns

6.1.1.5 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.

6.1.1.6 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
```

6.1.1.7 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>}
```

Alternatively, if your DataFrame contains columns that are not in the schema, and you would like these to be dropped on validation, you can specify `strict='filter'`.

```
import pandas as pd
import pandera as pa

from pandera import Column, DataFrameSchema

df = pd.DataFrame({"column1": ["drop", "me"],"column2": ["keep", "me"]})
schema = DataFrameSchema({"column2": Column(pa.String)}, strict='filter')
```

(continues on next page)

(continued from previous page)

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

```
   column2
0     keep
1       me
```

6.1.1.8 Validating the order of the columns

For some applications the order of the columns is important. For example:

- If you want to use selection by position instead of the more common selection by label.
- Machine learning: Many ML libraries will cast a Dataframe to numpy arrays, for which order becomes crucial.

To validate the order of the Dataframe columns, specify `ordered=True`:

```
import pandas as pd
import pandera as pa

schema = pa.DataFrameSchema(
    columns={"a": pa.Column(pa.Int), "b": pa.Column(pa.Int)}, ordered=True
)
df = pd.DataFrame({"b": [1], "a": [1]})
print(schema.validate(df))
```

```
Traceback (most recent call last):
...
SchemaError: column 'b' out-of-order
```

6.1.2 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
```

6.1.3 MultiIndex Validation

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

6.1.3.1 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
```

6.1.3.2 MultiIndex Indexes

The *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 ]],
        names=["index0", "index1"]
    )
)

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

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

6.1.4 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)
    },
)
```

(continues on next page)

(continued from previous page)

```
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
```

6.1.5 DataFrameSchema Transformations

Once you've defined a schema, you can then make modifications to it, both on the schema level – such as adding or removing columns and setting or resetting the index – or on the column level – such as changing the data type or checks.

This is useful for re-using schema objects in a data pipeline when additional computation has been done on a dataframe, where the column objects may have changed or perhaps where additional checks may be required.

```
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)
```

```
<Schema DataFrameSchema(
    columns={
        'col1': <Schema Column(name=col1, type=int)>
    },
    checks=[],
    coerce=False,
    pandas_dtype=None,
    index=None,
    strict=True
    name=None,
    ordered=False
)>
```

If during the course of a data pipeline one of your columns is moved into the index, you can simply update the initial input schema using the *set_index()* method to create a schema for the pipeline output.

```
import pandera as pa

from pandera import Column, DataFrameSchema, Check, Index

schema = DataFrameSchema(
    {
        "column1": Column(pa.Int),
        "column2": Column(pa.Float)
    },
    index=Index(pa.Int, name = "column3"),
    strict=True,
    coerce=True,
)
print(schema.set_index(["column1"], append = True))
```

```
<Schema DataFrameSchema(
    columns={
        'column2': <Schema Column(name=column2, type=float)>
    },
    checks=[],
    coerce=True,
    pandas_dtype=None,
    index=<Schema MultiIndex(
        indexes=[
            <Schema Index(name=column3, type=int)>
            <Schema Index(name=column1, type=int)>
```

(continues on next page)

(continued from previous page)

```
        ]
        coerce=False,
        strict=False,
        name=None,
        ordered=True
    )>,
    strict=True
    name=None,
    ordered=False
)>
```

The available methods for altering the schema are: *add_columns()*, *remove_columns()*, *update_columns()*, *rename_columns()*, *set_index()*, and *reset_index()*.

6.2 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
```

6.3 Checks

6.3.1 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)

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)
    ]),
})
```

6.3.2 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.

6.3.3 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),
```

(continues on next page)

(continued from previous page)

```
            # 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.

6.3.4 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.

6.3.5 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"),
```

(continues on next page)

(continued from previous page)

```
            # 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)),
    "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)
```

6.3.6 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)
})
```

(continues on next page)

(continued from previous page)

```
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),
    },
    # 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.

6.3.7 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),
```

(continues on next page)

(continued from previous page)

```
    "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,
)

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(name=var2, type=None)> failed series or dataframe validator 0:
<Check _hypothesis_check: normality test>
```

6.3.8 Registering Custom Checks

`pandera` now offers an interface to register custom checks functions so that they're available in the *`Check`* namespace. See *the extensions* document for more information.

6.4 Hypothesis Testing

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

Note: The hypothesis feature requires a pandera installation with `hypotheses` dependency set. See the *installation* instructions for more details.

6.4.1 Overview

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):
```

(continues on next page)

(continued from previous page)

```
    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)
```

6.4.2 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

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)
```

6.5 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.

6.5.1 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.
```

(continues on next page)

(continued from previous page)

```
@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):
    ...
```

6.5.2 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
```

(continues on next page)

(continued from previous page)

```
# 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)
```

6.5.3 Check IO

For convenience, you can also use the *check_io()* decorator where you can specify input and output schemas more concisely:

```
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(int),
   "column2": Column(float),
})

out_schema = in_schema.add_columns({"column3": Column(float)})

@pa.check_io(df1=in_schema, df2=in_schema, out=out_schema)
def preprocessor(df1, df2):
    return (df1 + df2).assign(column3=lambda x: x.column1 + x.column2)

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

```
   column1  column2  column3
0        2     -2.6     -0.6
1        8     -2.8      5.2
2        0     -5.8     -5.8
3       20    -20.2     -0.2
4       18    -40.8    -22.8
```

6.6 Schema Inference

New in version 0.4.0

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)
```

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

- *add_columns()*
- *remove_columns()*
- *update_column()*

For *SeriesSchema* objects:

- `set_checks()`

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

6.6.1 Schema Persistence

The schema persistence feature requires a pandera installation with the `io` extension. See the *installation* instructions for more details.

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

6.6.1.1 Write to a Python script

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

```
# 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 = schema.to_script()
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=[
```

(continues on next page)

(continued from previous page)

```
                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")
                ),
            ],
            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.

6.6.1.2 Write to YAML

You can also write the schema object to a yaml file with *to_yaml()*, and you can then read it into memory with *from_yaml()*. The *to_yaml()* and *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: 0.6.5
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
```

(continues on next page)

(continued from previous page)

```
  column2:
    pandas_dtype: str
    nullable: false
    checks: null
    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
checks: null
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.

6.7 Schema Models

new in 0.5.0

`pandera` provides a class-based API that's heavily inspired by pydantic. In contrast to the *object-based API*, you can define schema models in much the same way you'd define `pydantic` models.

Schema Models are annotated with the *`pandera.typing`* module using the standard typing syntax. Models can be explictly converted to a *`DataFrameSchema`* or used to validate a `DataFrame` directly.

Note: Due to current limitations in the pandas library (see discussion here), `pandera` annotations are only used for **run-time** validation and **cannot** be leveraged by static-type checkers like mypy. See the discussion here for more details.

6.7.1 Basic Usage

```
import pandas as pd
import pandera as pa
from pandera.typing import Index, DataFrame, Series


class InputSchema(pa.SchemaModel):
    year: Series[int] = pa.Field(gt=2000, coerce=True)
    month: Series[int] = pa.Field(ge=1, le=12, coerce=True)
    day: Series[int] = pa.Field(ge=0, le=365, coerce=True)

class OutputSchema(InputSchema):
    revenue: Series[float]

@pa.check_types
def transform(df: DataFrame[InputSchema]) -> DataFrame[OutputSchema]:
    return df.assign(revenue=100.0)


df = pd.DataFrame({
    "year": ["2001", "2002", "2003"],
    "month": ["3", "6", "12"],
    "day": ["200", "156", "365"],
})

transform(df)

invalid_df = pd.DataFrame({
    "year": ["2001", "2002", "1999"],
    "month": ["3", "6", "12"],
    "day": ["200", "156", "365"],
})
transform(invalid_df)
```

```
Traceback (most recent call last):
...
pandera.errors.SchemaError: <Schema Column: 'year' type=<class 'int'>> failed element-
↪wise validator 0:
<Check greater_than: greater_than(2000)>
failure cases:
   index  failure_case
0      2          1999
```

As you can see in the example above, you can define a schema by sub-classing *SchemaModel* and defining column/index fields as class attributes. The *check_types()* decorator is required to perform validation of the dataframe at run-time.

Note that *Field* s apply to both *Column* and *Index* objects, exposing the built-in `Check` s via key-word arguments.

(New in 0.6.2) When you access a class attribute defined on the schema, it will return the name of the column used in the validated *pd.DataFrame*. In the example above, this will simply be the string *"year"*.

```
print(f"Column name for 'year' is {InputSchema.year}\n")
print(df.loc[:, [InputSchema.year, "day"]])
```

```
Column name for 'year' is year

   year  day
0  2001  200
1  2002  156
2  2003  365
```

6.7.2 Converting to DataFrameSchema

You can easily convert a *SchemaModel* class into a *DataFrameSchema*:

```
print(InputSchema.to_schema())
```

```
<Schema DataFrameSchema(
    columns={
        'year': <Schema Column(name=year, type=<class 'int'>)>
        'month': <Schema Column(name=month, type=<class 'int'>)>
        'day': <Schema Column(name=day, type=<class 'int'>)>
    },
    checks=[],
    coerce=False,
    pandas_dtype=None,
    index=None,
    strict=False
    name=None,
    ordered=False
)>
```

Or use the *validate()* method to validate dataframes:

```
print(InputSchema.validate(df))
```

```
   year  month  day
0  2001      3  200
1  2002      6  156
2  2003     12  365
```

6.7.3 Excluded attributes

Class variables which begin with an underscore will be automatically excluded from the model. *Config* is also a reserved name. However, *aliases* can be used to circumvent these limitations.

6.7.4 Supported dtypes

Any dtypes supported by `pandera` can be used as type parameters for `Series` and `Index`. There are, however, a couple of gotchas.

6.7.4.1 Dtype aliases

The enumeration *PandasDtype* is not directly supported because the type parameter of a typing.Generic cannot be an enumeration[1]. Instead, you can use the *pandera.typing* counterparts: pandera.typing.Category, pandera.typing.Float32, ...

✓ Good:

```
import pandera as pa
from pandera.typing import Series, String

class Schema(pa.SchemaModel):
    a: Series[String]
```

Bad:

```
class Schema(pa.SchemaModel):
    a: Series[pa.PandasDtype.String]
```

```
Traceback (most recent call last):
...
TypeError: python type '<class 'typing.Generic'>' not recognized as pandas data type
```

6.7.4.2 Type Vs instance

You must give a **type**, not an **instance**.

✓ Good:

```
import pandas as pd

class Schema(pa.SchemaModel):
    a: Series[pd.StringDtype]
```

Bad:

```
class Schema(pa.SchemaModel):
    a: Series[pd.StringDtype()]
```

```
Traceback (most recent call last):
...
TypeError: Parameters to generic types must be types. Got StringDtype.
```

[1] It is actually possible to use a PandasDtype by encasing it in a typing.Literal like Series[Literal[PandasDtype.Category]]. *pandera.typing* defines aliases to reduce boilerplate.

6.7.4.3 Parametrized dtypes

Pandas supports a couple of parametrized dtypes. As of pandas 1.2.0:

Kind of Data	Data Type	Parameters
tz-aware datetime	`DatetimeTZDtype`	`unit`, `tz`
Categorical	`CategoricalDtype`	`categories`, `ordered`
period	`PeriodDtype`	`freq`
sparse	`SparseDtype`	`dtype`, `fill_value`
intervals	`IntervalDtype`	`subtype`

6.7.4.3.1 Annotated

Parameters can be given via `typing.Annotated`. It requires python > 3.9 or typing_extensions, which is already a requirement of Pandera. Unfortunately `typing.Annotated` has not been backported to python 3.6.

✓ Good:

```
try:
    from typing import Annotated  # python 3.9+
except ImportError:
    from typing_extensions import Annotated

class Schema(pa.SchemaModel):
    col: Series[Annotated[pd.DatetimeTZDtype, "ns", "est"]]
```

Furthermore, you must pass all parameters in the order defined in the dtype's constructor (see *table*).

Bad:

```
class Schema(pa.SchemaModel):
    col: Series[Annotated[pd.DatetimeTZDtype, "utc"]]

Schema.to_schema()
```

```
Traceback (most recent call last):
...
TypeError: Annotation 'DatetimeTZDtype' requires all positional arguments ['unit', 'tz
↪'].
```

6.7.4.3.2 Field

✓ Good:

```
class SchemaFieldDatetimeTZDtype(pa.SchemaModel):
    col: Series[pd.DatetimeTZDtype] = pa.Field(dtype_kwargs={"unit": "ns", "tz": "EST
↪"})
```

You cannot use both `typing.Annotated` and `dtype_kwargs`.

Bad:

```
class SchemaFieldDatetimeTZDtype(pa.SchemaModel):
    col: Series[Annotated[pd.DatetimeTZDtype, "ns", "est"]] = pa.Field(dtype_kwargs={
↪"unit": "ns", "tz": "EST"})

Schema.to_schema()
```

```
Traceback (most recent call last):
...
TypeError: Cannot specify redundant 'dtype_kwargs' for pandera.typing.Series[typing_
↪extensions.Annotated[pandas.core.dtypes.dtypes.DatetimeTZDtype, 'ns', 'est']].
Usage Tip: Drop 'typing.Annotated'.
```

6.7.5 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, annotate it with `typing.Optional`.

```
from typing import Optional

import pandas as pd
import pandera as pa
from pandera.typing import Series


class Schema(pa.SchemaModel):
    a: Series[str]
    b: Optional[Series[int]]


df = pd.DataFrame({"a": ["2001", "2002", "2003"]})
Schema.validate(df)
```

6.7.6 Schema Inheritance

You can also use inheritance to build schemas on top of a base schema.

```
class BaseSchema(pa.SchemaModel):
    year: Series[str]

class FinalSchema(BaseSchema):
    year: Series[int] = pa.Field(ge=2000, coerce=True)  # overwrite the base type
    passengers: Series[int]
    idx: Index[int] = pa.Field(ge=0)

df = pd.DataFrame({
    "year": ["2000", "2001", "2002"],
})

@pa.check_types
def transform(df: DataFrame[BaseSchema]) -> DataFrame[FinalSchema]:
    return (
        df.assign(passengers=[61000, 50000, 45000])
        .set_index(pd.Index([1, 2, 3]))
```

(continues on next page)

(continued from previous page)

```
        .astype({"year": int})
    )

print(transform(df))
```

```
   year  passengers
1  2000       61000
2  2001       50000
3  2002       45000
```

6.7.7 Config

Schema-wide options can be controlled via the `Config` class on the `SchemaModel` subclass. The full set of options can be found in the *`BaseConfig`* class.

```
class Schema(pa.SchemaModel):

    year: Series[int] = pa.Field(gt=2000, coerce=True)
    month: Series[int] = pa.Field(ge=1, le=12, coerce=True)
    day: Series[int] = pa.Field(ge=0, le=365, coerce=True)

    class Config:
        name = "BaseSchema"
        strict = True
        coerce = True
        foo = "bar"  # Interpreted as dataframe check
```

It is not required for the `Config` to subclass *`BaseConfig`* but it **must** be named '**Config**'.

See *Registered Custom Checks with the Class-based API* for details on using registered dataframe checks.

6.7.8 MultiIndex

The *`MultiIndex`* capabilities are also supported with the class-based API:

```
import pandera as pa
from pandera.typing import Index, Series

class MultiIndexSchema(pa.SchemaModel):

    year: Index[int] = pa.Field(gt=2000, coerce=True)
    month: Index[int] = pa.Field(ge=1, le=12, coerce=True)
    passengers: Series[int]

    class Config:
        # provide multi index options in the config
        multiindex_name = "time"
        multiindex_strict = True
        multiindex_coerce = True

index = MultiIndexSchema.to_schema().index
print(index)
```

```
<Schema MultiIndex(
    indexes=[
        <Schema Index(name=year, type=<class 'int'>)>
        <Schema Index(name=month, type=<class 'int'>)>
    ]
    coerce=True,
    strict=True,
    name=time,
    ordered=True
)>
```

```
from pprint import pprint

pprint({name: col.checks for name, col in index.columns.items()})
```

```
{'month': [<Check greater_than_or_equal_to: greater_than_or_equal_to(1)>,
        <Check less_than_or_equal_to: less_than_or_equal_to(12)>],
'year': [<Check greater_than: greater_than(2000)>]}
```

Multiple `Index` annotations are automatically converted into a *`MultiIndex`*. MultiIndex options are given in the *Config*.

6.7.9 Custom Checks

Unlike the object-based API, custom checks can be specified as class methods.

6.7.9.1 Column/Index checks

```
import pandera as pa
from pandera.typing import Index, Series

class CustomCheckSchema(pa.SchemaModel):

    a: Series[int] = pa.Field(gt=0, coerce=True)
    abc: Series[int]
    idx: Index[str]

    @pa.check("a", name="foobar")
    def custom_check(cls, a: Series[int]) -> Series[bool]:
        return a < 100

    @pa.check("^a", regex=True, name="foobar")
    def custom_check_regex(cls, a: Series[int]) -> Series[bool]:
        return a > 0

    @pa.check("idx")
    def check_idx(cls, idx: Index[int]) -> Series[bool]:
        return idx.str.contains("dog")
```

Note:

- You can supply the key-word arguments of the *`Check`* class initializer to get the flexibility of *groupby checks*
- Similarly to `pydantic`, `classmethod()` decorator is added behind the scenes if omitted.

- You still may need to add the `@classmethod` decorator *after* the *check()* decorator if your static-type checker or linter complains.
- Since `checks` are class methods, the first argument value they receive is a SchemaModel subclass, not an instance of a model.

```
from typing import Dict

class GroupbyCheckSchema(pa.SchemaModel):

    value: Series[int] = pa.Field(gt=0, coerce=True)
    group: Series[str] = pa.Field(isin=["A", "B"])

    @pa.check("value", groupby="group", regex=True, name="check_means")
    def check_groupby(cls, grouped_value: Dict[str, Series[int]]) -> bool:
        return grouped_value["A"].mean() < grouped_value["B"].mean()

df = pd.DataFrame({
    "value": [100, 110, 120, 10, 11, 12],
    "group": list("AAABBB"),
})

print(GroupbyCheckSchema.validate(df))
```

```
Traceback (most recent call last):
...
pandera.errors.SchemaError: <Schema Column: 'value' type=<class 'int'>> failed series␣
↪validator 1:
<Check check_means>
```

6.7.9.2 DataFrame Checks

You can also define dataframe-level checks, similar to the *object-based API*, using the `dataframe_check()` decorator:

```
import pandas as pd
import pandera as pa
from pandera.typing import Index, Series

class DataFrameCheckSchema(pa.SchemaModel):

    col1: Series[int] = pa.Field(gt=0, coerce=True)
    col2: Series[float] = pa.Field(gt=0, coerce=True)
    col3: Series[float] = pa.Field(lt=0, coerce=True)

    @pa.dataframe_check
    def product_is_negative(cls, df: pd.DataFrame) -> Series[bool]:
        return df["col1"] * df["col2"] * df["col3"] < 0

df = pd.DataFrame({
    "col1": [1, 2, 3],
    "col2": [5, 6, 7],
    "col3": [-1, -2, -3],
})

DataFrameCheckSchema.validate(df)
```

6.7.9.3 Inheritance

The custom checks are inherited and therefore can be overwritten by the subclass.

```
import pandas as pd
import pandera as pa
from pandera.typing import Index, Series

class Parent(pa.SchemaModel):

    a: Series[int] = pa.Field(coerce=True)

    @pa.check("a", name="foobar")
    def check_a(cls, a: Series[int]) -> Series[bool]:
        return a < 100


class Child(Parent):

    a: Series[int] = pa.Field(coerce=False)

    @pa.check("a", name="foobar")
    def check_a(cls, a: Series[int]) -> Series[bool]:
        return a > 100

is_a_coerce = Child.to_schema().columns["a"].coerce
print(f"coerce: {is_a_coerce}")
```

```
coerce: False
```

```
df = pd.DataFrame({"a": [1, 2, 3]})
print(Child.validate(df))
```

```
Traceback (most recent call last):
...
pandera.errors.SchemaError: <Schema Column: 'a' type=<class 'int'>> failed element-
↪wise validator 0:
<Check foobar>
failure cases:
    index  failure_case
0       0             1
1       1             2
2       2             3
```

6.7.10 Aliases

SchemaModel supports columns which are not valid python variable names via the argument *alias* of *Field*.

Checks must reference the aliased names.

```
import pandera as pa
import pandas as pd

class Schema(pa.SchemaModel):
    col_2020: pa.typing.Series[int] = pa.Field(alias=2020)
```

(continues on next page)

(continued from previous page)

```
    idx: pa.typing.Index[int] = pa.Field(alias="_idx", check_name=True)

    @pa.check(2020)
    def int_column_lt_100(cls, series):
        return series < 100


df = pd.DataFrame({2020: [99]}, index=[0])
df.index.name = "_idx"

print(Schema.validate(df))
```

```
      2020
_idx
0       99
```

(New in 0.6.2) The *alias* is respected when using the class attribute to get the underlying *pd.DataFrame* column name or index level name.

```
print(Schema.col_2020)
```

```
2020
```

Very similar to the example above, you can also use the variable name directly within the class scope, and it will respect the alias.

Note: To access a variable from the class scope, you need to make it a class attribute, and therefore assign it a default *Field*.

```
import pandera as pa
import pandas as pd

class Schema(pa.SchemaModel):
    a: pa.typing.Series[int] = pa.Field()
    col_2020: pa.typing.Series[int] = pa.Field(alias=2020)

    @pa.check(col_2020)
    def int_column_lt_100(cls, series):
        return series < 100

    @pa.check(a)
    def int_column_gt_100(cls, series):
        return series > 100


df = pd.DataFrame({2020: [99], "a": [101]})
print(Schema.validate(df))
```

```
      2020    a
0       99  101
```

6.7.11 Footnotes

6.8 Lazy Validation

New in version 0.4.0

By default, when you call the `validate` method on schema or schema component objects, a *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

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"],
```

(continues on next page)

(continued from previous page)

```
    "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
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.failure_cases # dataframe of schema errors
    err.data # invalid dataframe
```
```

As you can see from the output above, a *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.failure_cases)
    print("\nDataFrame object that failed validation:")
```

(continues on next page)

(continued from previous page)

```
    print(err.data)
```

```
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  float_column          greater_than(0)            0
5           Column    str_column              equal_to(a)            0
6           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               0     0
5               b     1
6               d     2

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
```

6.9 Data Synthesis Strategies (new)

new in 0.6.0

`pandera` provides a utility for generating synthetic data purely from pandera schema or schema component objects. Under the hood, the schema metadata is collected to create a data-generating strategy using hypothesis, which is a property-based testing library.

6.9.1 Basic Usage

Once you've defined a schema, it's easy to generate examples:

```
import pandera as pa

schema = pa.DataFrameSchema(
    {
        "column1": pa.Column(int, pa.Check.eq(10)),
        "column2": pa.Column(float, pa.Check.eq(0.25)),
        "column3": pa.Column(str, pa.Check.eq("foo")),
    }
)
print(schema.example(size=3))
```

```
    column1  column2 column3
 0       10     0.25     foo
```

(continues on next page)

(continued from previous page)

```
1       10      0.25      foo
2       10      0.25      foo
```

Note that here we've constrained the specific values in each column using *Check* s in order to make the data generation process deterministic for documentation purposes.

6.9.2 Usage in Unit Tests

The `example` method is available for all schemas and schema components, and is primarily meant to be used interactively. It *could* be used in a script to generate test cases, but `hypothesis` recommends against doing this and instead using the `strategy` method to create a `hypothesis` strategy that can be used in `pytest` unit tests.

```
import hypothesis

def processing_fn(df):
    return df.assign(column4=df.column1 * df.column2)

@hypothesis.given(schema.strategy(size=5))
def test_processing_fn(dataframe):
    result = processing_fn(dataframe)
    assert "column4" in result
```

The above example is trivial, but you get the idea! Schema objects can create a `strategy` that can then be collected by a pytest runner. We could also run the tests explicitly ourselves, or run it as a `unittest.TestCase`. For more information on testing with hypothesis, see the hypothesis quick start guide.

A more practical example involves using *schema transformations*. We can modify the function above to make sure that `processing_fn` actually outputs the correct result:

```
out_schema = schema.add_columns({"column4": pa.Column(float)})

@pa.check_output(out_schema)
def processing_fn(df):
    return df.assign(column4=df.column1 * df.column2)

@hypothesis.given(schema.strategy(size=5))
def test_processing_fn(dataframe):
    processing_fn(dataframe)
```

Now the `test_processing_fn` simply becomes an execution test, raising a *SchemaError* if `processing_fn` doesn't add `column4` to the dataframe.

6.9.3 Strategies and Examples from Schema Models

You can also use the *class-based API* to generate examples. Here's the equivalent schema model for the above examples:

```
from pandera.typing import Series, DataFrame

class InSchema(pa.SchemaModel):
    column1: Series[int] = pa.Field(eq=10)
    column2: Series[float] = pa.Field(eq=0.25)
    column3: Series[str] = pa.Field(eq="foo")
```

(continues on next page)

(continued from previous page)

```
class OutSchema(InSchema):
    column4: Series[float]

@pa.check_types
def processing_fn(df: DataFrame[InSchema]) -> DataFrame[OutSchema]:
    return df.assign(column4=df.column1 * df.column2)

@hypothesis.given(InSchema.strategy(size=5))
def test_processing_fn(dataframe):
    processing_fn(dataframe)
```

6.9.4 Checks as Constraints

As you may have noticed in the first example, *Check* s further constrain the data synthesized from a strategy. Without checks, the `example` method would simply generate any value of the specified type. You can specify multiple checks on a column and `pandera` should be able to generate valid data under those constraints.

```
schema_multiple_checks = pa.DataFrameSchema({
    "column1": pa.Column(
        float, checks=[
            pa.Check.gt(0),
            pa.Check.lt(1e10),
            pa.Check.notin([-100, -10, 0]),
        ]
    )
})

for _ in range(100):
    # generate 10 rows of the dataframe
    sample_data = schema_multiple_checks.example(size=10)

    # validate the sampled data
    schema_multiple_checks(sample_data)
```

One caveat here is that it's up to you to define a set of checks that are jointly satisfiable. If not, an `Unsatisfiable` exception will be raised:

```
schema_multiple_checks = pa.DataFrameSchema({
    "column1": pa.Column(
        float, checks=[
            # nonsensical constraints
            pa.Check.gt(0),
            pa.Check.lt(-10),
        ]
    )
})

schema_multiple_checks.example(size=10)
```

```
Traceback (most recent call last):
...
Unsatisfiable: Unable to satisfy assumptions of hypothesis example_generating_inner_
↪function.
```

6.9.4.1 Check Strategy Chaining

If you specify multiple checks for a particular column, this is what happens under the hood:

- The first check in the list is the *base strategy*, which `hypothesis` uses to generate data.
- All subsequent checks filter the values generated by the previous strategy such that it fulfills the constraints of current check.

To optimize efficiency of the data-generation procedure, make sure to specify the most restrictive constraint of a column as the *base strategy* and build other constraints on top of it.

6.9.4.2 In-line Custom Checks

One of the strengths of `pandera` is its flexibility with regard to defining custom checks on the fly:

```
schema_inline_check = pa.DataFrameSchema({
    "col": pa.Column(str, pa.Check(lambda s: s.isin({"foo", "bar"})))
})
```

One of the disadvantages of this is that the fallback strategy is to simply apply the check to the generated data, which can be highly inefficient. In this case, `hypothesis` will generate strings and try to find examples of strings that are in the set `{"foo", "bar"}`, which will be very slow and most likely raise an `Unsatisfiable` exception. To get around this limitation, you can register custom checks and define strategies that correspond to them.

6.9.5 Defining Custom Strategies

All built-in *Check* s are associated with a data synthesis strategy. You can define your own data synthesis strategies by using the *extensions API* to register a custom check function with a corresponding strategy.

6.10 Extensions (new)

new in 0.6.0

6.10.1 Registering Custom Check Methods

One of the strengths of `pandera` is its flexibility in enabling you to defining in-line custom checks on the fly:

```
import pandera as pa

# checks elements in a column/dataframe
element_wise_check = pa.Check(lambda x: x < 0, element_wise=True)

# applies the check function to a dataframe/series
vectorized_check = pa.Check(lambda series_or_df: series_or_df < 0)
```

However, there are two main disadvantages of schemas with inline custom checks:

1. they are not serializable with the *IO interface*.
2. you can't use them to *synthesize data* because the checks are not associated with a `hypothesis` strategy.

`pandera` now offers a way to register custom checks so that they're available in the *Check* class as a check method. Here let's define a custom method that checks whether a pandas object contains elements that lie within two values.

```
import pandera as pa
import pandera.extensions as extensions
import pandas as pd

@extensions.register_check_method(statistics=["min_value", "max_value"])
def is_between(pandas_obj, *, min_value, max_value):
    return (min_value <= pandas_obj) & (pandas_obj <= max_value)

schema = pa.DataFrameSchema({
    "col": pa.Column(int, pa.Check.is_between(min_value=1, max_value=10))
})

data = pd.DataFrame({"col": [1, 5, 10]})
print(schema(data))
```

```
   col
0    1
1    5
2   10
```

As you can see, a custom check's first argument is a pandas series or dataframe by default (more on that later), followed by keyword-only arguments, specified with the * syntax.

The *register_check_method()* requires you to explicitly name the check `statistics` via the keyword argument, which are essentially the constraints placed by the check on the pandas data structure.

6.10.2 Specifying a Check Strategy

To specify a check strategy with your custom check, you'll need to install the *strategies extension*. First let's look at a trivially simple example, where the check verifies whether a column is equal to a certain value:

```
def custom_equals(pandas_obj, *, value):
    return pandas_obj == value
```

The corresponding strategy for this check would be:

```
from typing import Optional
import hypothesis
import pandera.strategies as st

def equals_strategy(
    pandas_dtype: pa.PandasDtype,
    strategy: Optional[st.SearchStrategy] = None,
    *,
    value,
):
    if strategy is None:
        return st.pandas_dtype_strategy(
            pandas_dtype, strategy=hypothesis.strategies.just(value),
        )
    return strategy.filter(lambda x: x == value)
```

As you may notice, the `pandera` strategy interface is has two arguments followed by keyword-only arguments that match the check function keyword-only check statistics. The `pandas_dtype` positional argument is useful for ensuring the correct data type. In the above example, we're using the *pandas_dtype_strategy()* strategy to make sure the generated `value` is of the correct data type.

The optional `strategy` argument allows us to use the check strategy as a *base strategy* or a *chained strategy*. There's a detail that we're responsible for implementing in the strategy function body: we need to handle two cases to account for *strategy chaining*:

1. when the strategy function is being used as a *base strategy*, i.e. when `strategy` is `None`
2. when the strategy function is being chained from a previously-defined strategy, i.e. when `strategy` is not `None`.

Finally, to register the custom check with the strategy, use the *`register_check_method()`* decorator:

```
@extensions.register_check_method(
    statistics=["value"], strategy=equals_strategy
)
def custom_equals(pandas_obj, *, value):
    return pandas_obj == value
```

Let's unpack what's going in here. The `custom_equals` function only has a single statistic, which is the `value` argument, which we've also specified in *`register_check_method()`*. This means that the associated check strategy must match its keyword-only arguments.

Going back to our `is_between` function example, here's what the strategy would look like:

```
def in_between_strategy(
    pandas_dtype: pa.PandasDtype,
    strategy: Optional[st.SearchStrategy] = None,
    *,
    min_value,
    max_value
):
    if strategy is None:
        return st.pandas_dtype_strategy(
            pandas_dtype,
            min_value=min_value,
            max_value=max_value,
            exclude_min=False,
            exclude_max=False,
        )
    return strategy.filter(lambda x: min_value <= x <= max_value)

@extensions.register_check_method(
    statistics=["min_value", "max_value"],
    strategy=in_between_strategy,
)
def is_between_with_strat(pandas_obj, *, min_value, max_value):
    return (min_value <= pandas_obj) & (pandas_obj <= max_value)
```

6.10.3 Check Types

The extensions module also supports registering *element-wise* and *groupby* checks.

6.10.3.1 Element-wise Checks

```
@extensions.register_check_method(
    statistics=["val"],
    check_type="element_wise",
)
def element_wise_equal_check(element, *, val):
    return element == val
```

Note that the first argument of `element_wise_equal_check` is a single element in the column or dataframe.

6.10.3.2 Groupby Checks

In this groupby check, we're verifying that the values of one column for `group_a` are, on average, greater than those of `group_b`:

```
from typing import Dict

@extensions.register_check_method(
    statistics=["group_a", "group_b"],
    check_type="groupby",
)
def groupby_check(dict_groups: Dict[str, pd.Series], *, group_a, group_b):
    return dict_groups[group_a].mean() > dict_groups[group_b].mean()

data = pd.DataFrame({
    "values": [20, 10, 1, 15],
    "groups": list("xxyy"),
})

schema = pa.DataFrameSchema({
    "values": pa.Column(
        int,
        pa.Check.groupby_check(group_a="x", group_b="y", groupby="groups"),
    ),
    "groups": pa.Column(str),
})

print(schema(data))
```

```
   values groups
0      20      x
1      10      x
2       1      y
3      15      y
```

6.10.4 Registered Custom Checks with the Class-based API

Since registered checks are part of the *Check* namespace, you can also use custom checks with the *class-based API*:

```
from pandera.typing import Series

class Schema(pa.SchemaModel):
    col1: Series[str] = pa.Field(custom_equals="value")
    col2: Series[int] = pa.Field(is_between={"min_value": 0, "max_value": 10})

data = pd.DataFrame({
    "col1": ["value"] * 5,
    "col2": range(5)
})

print(Schema.validate(data))
```

```
    col1  col2
0  value     0
1  value     1
2  value     2
3  value     3
4  value     4
```

DataFrame checks can be attached by using the *Config* class. Any field names that do not conflict with existing fields of *BaseConfig* and do not start with an underscore (_) are interpreted as the name of registered checks. If the value is a tuple or dict, it is interpreted as the positional or keyword arguments of the check, and as the first argument otherwise.

For example, to register zero, one, and two statistic dataframe checks one could do the following:

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


@extensions.register_check_method()
def is_small(df):
    return sum(df.shape) < 1000


@extensions.register_check_method(statistics=["fraction"])
def total_missing_fraction_less_than(df, *, fraction: float):
    return (1 - df.count().sum().item() / sum(df.shape)) < fraction


@extensions.register_check_method(statistics=["col_a", "colb"])
def col_mean_a_greater_than_b(df, *, col_a: str, col_b: str):
    return df[col_a].mean() > df[col_b].mean()


from pandera.typing import Series


class Schema(pa.SchemaModel):
    col1: Series[float] = pa.Field(nullable=True, ignore_na=False)
```

(continues on next page)

(continued from previous page)

```
    col2: Series[float] = pa.Field(nullable=True, ignore_na=False)

    class Config:
        is_small = ()
        total_missing_fraction_less_than = 0.6
        col_mean_a_greater_than_b = {"col_a": "col2", "col_b": "col1"}


data = pd.DataFrame({
    "col1": [float('nan')] * 3 + [0.5, 0.3, 0.1],
    "col2": np.arange(6.),
})

print(Schema.validate(data))
```

```
   col1  col2
0   NaN   0.0
1   NaN   1.0
2   NaN   2.0
3   0.5   3.0
4   0.3   4.0
5   0.1   5.0
```

6.11 API

The `io` module and built-in `Hypothesis` checks require a pandera installation with the corresponding extension, see the *installation* instructions for more details.

6.11.1 Schemas

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

6.11.1.1 pandera.schemas.DataFrameSchema

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

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.

- **pandas_dtype** (Union[str, type, *PandasDtype*, ExtensionDtype, dtype, None]) – datatype of the dataframe. This overrides the data types specified in any of the columns. If a string is specified, then assumes one of the valid pandas string values: http://pandas.pydata.org/pandas-docs/stable/basics.html#dtypes.
- **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. This has no effect on columns where `pandas_dtype=None`
- **strict** (Union[bool, str]) – ensure that all and only the columns defined in the schema are present in the dataframe. If set to 'filter', only the columns in the schema will be passed to the validated dataframe. If set to filter and columns defined in the schema are not present in the dataframe, will throw an error.
- **name** (Optional[str]) – name of the schema.
- **ordered** (bool) – whether or not to validate the columns order.

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.

```
>>> 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.

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.
`ordered`	Whether or not to validate the columns order.
`pandas_dtype`	Get the pandas dtype property.
`pdtype`	PandasDtype of the dataframe.

Methods

`__init__`	Initialize DataFrameSchema validator.
`add_columns`	Create a copy of the *`DataFrameSchema`* with extra columns.
`coerce_dtype`	Coerce dataframe to the type specified in pandas_dtype.
`example`	Generate an example of a particular size.
`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.
`reset_index`	A method for resetting the `Index` of a *`DataFrameSchema`*
`select_columns`	Select subset of columns in the schema.
`set_index`	A method for setting the `Index` of a *`DataFrameSchema`*, via an existing `Column` or list of columns.
`strategy`	Create a `hypothesis` strategy for generating a DataFrame.
`to_script`	Create DataFrameSchema from yaml file.
`to_yaml`	Write DataFrameSchema to yaml file.
`update_column`	Create copy of a *`DataFrameSchema`* with updated column properties.
`update_columns`	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.

6.11.1.1.1 pandera.schemas.DataFrameSchema.__init__

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

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.
- **pandas_dtype** (Union[str, type, *PandasDtype*, ExtensionDtype, dtype, None]) – datatype of the dataframe. This overrides the data types specified in any of the columns. If a string is specified, then assumes one of the valid pandas string values: http://pandas.pydata.org/pandas-docs/stable/basics.html#dtypes.
- **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. This has no effect on columns where pandas_dtype=None
- **strict** (Union[bool, str]) – ensure that all and only the columns defined in the schema are present in the dataframe. If set to 'filter', only the columns in the schema will be passed to the validated dataframe. If set to filter and columns defined in the schema are not present in the dataframe, will throw an error.
- **name** (Optional[str]) – name of the schema.
- **ordered** (bool) – whether or not to validate the columns order.

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.

```
>>> 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.
```

(continues on next page)

(continued from previous page)

```
...     "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.

6.11.1.1.2 pandera.schemas.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.

Example

To add columns to the schema, pass a dictionary with column name and Column instance key-value pairs.

```
>>> import pandera as pa
>>>
>>> example_schema = pa.DataFrameSchema(
...     {
...         "category": pa.Column(pa.String),
...         "probability": pa.Column(pa.Float),
...     }
... )
>>> print(
...     example_schema.add_columns({"even_number": pa.Column(pa.Bool)})
... )
<Schema DataFrameSchema(
    columns={
        'category': <Schema Column(name=category, type=str)>
        'probability': <Schema Column(name=probability, type=float)>
        'even_number': <Schema Column(name=even_number, type=bool)>
    },
    checks=[],
    coerce=False,
    pandas_dtype=None,
    index=None,
    strict=False
    name=None,
    ordered=False
)>
```

See also:

remove_columns()

6.11.1.1.3 pandera.schemas.DataFrameSchema.coerce_dtype

DataFrameSchema.**coerce_dtype**(*obj*)

Coerce dataframe to the type specified in pandas_dtype.

Parameters **obj** (DataFrame) – dataframe to coerce.

Return type DataFrame

Returns dataframe with coerced dtypes

6.11.1.1.4 pandera.schemas.DataFrameSchema.example

DataFrameSchema.**example**(*size=None*, *n_regex_columns=1*)

Generate an example of a particular size.

Parameters **size** – number of elements in the generated DataFrame.

Return type DataFrame

Returns pandas DataFrame object.

6.11.1.1.5 pandera.schemas.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.

6.11.1.1.6 pandera.schemas.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.

6.11.1.1.7 pandera.schemas.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

Raises *SchemaInitError*: if column not in schema.

Example

To remove a column or set of columns from a schema, pass a list of columns to be removed:

```
>>> import pandera as pa
>>>
>>> example_schema = pa.DataFrameSchema(
...     {
...         "category" : pa.Column(pa.String),
...         "probability": pa.Column(pa.Float)
...     }
... )
>>>
>>> print(example_schema.remove_columns(["category"]))
<Schema DataFrameSchema(
    columns={
        'probability': <Schema Column(name=probability, type=float)>
    },
    checks=[],
    coerce=False,
    pandas_dtype=None,
    index=None,
    strict=False
    name=None,
    ordered=False
)>
```

See also:

add_columns()

6.11.1.1.8 pandera.schemas.DataFrameSchema.rename_columns

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

Rename columns using a dictionary of key-value pairs.

Parameters **rename_dict** (Dict[str, str]) – dictionary of 'old_name': 'new_name' key-value pairs.

Return type *DataFrameSchema*

Returns *DataFrameSchema* (copy of original)

Raises *SchemaInitError* if column not in the schema.

Example

To rename a column or set of columns, pass a dictionary of old column names and new column names, similar to the pandas DataFrame method.

```
>>> import pandera as pa
>>>
>>> example_schema = pa.DataFrameSchema({
...     "category" : pa.Column(pa.String),
...     "probability": pa.Column(pa.Float)
... })
>>>
>>> print(
...     example_schema.rename_columns({
...         "category": "categories",
...         "probability": "probabilities"
```

(continues on next page)

(continued from previous page)

```
...     })
... )
<Schema DataFrameSchema(
    columns={
        'categories': <Schema Column(name=categories, type=str)>
        'probabilities': <Schema Column(name=probabilities, type=float)>
    },
    checks=[],
    coerce=False,
    pandas_dtype=None,
    index=None,
    strict=False
    name=None,
    ordered=False
)>
```

See also:

update_column()

6.11.1.1.9 pandera.schemas.DataFrameSchema.reset_index

DataFrameSchema.**reset_index**(*level=None*, *drop=False*)

A method for resetting the Index of a *DataFrameSchema*

Parameters

- **level** (Optional[List[str]]) – list of labels
- **drop** (bool) – bool, default True

Return type *DataFrameSchema*

Returns a new *DataFrameSchema* with specified column(s) in the index.

Raises *SchemaInitError* if no index set in schema.

Examples

Similar to the pandas reset_index method on a pandas DataFrame, this method can be used to to fully or partially reset indices of a schema.

To remove the entire index from the schema, just call the reset_index method with default parameters.

```
>>> import pandera as pa
>>>
>>> example_schema = pa.DataFrameSchema(
...     {"probability" : pa.Column(pa.Float)},
...     index = pa.Index(name="unique_id", pandas_dtype=pa.Int)
... )
>>>
>>> print(example_schema.reset_index())
<Schema DataFrameSchema(
    columns={
        'probability': <Schema Column(name=probability, type=float)>
        'unique_id': <Schema Column(name=unique_id, type=int64)>
    },
    checks=[],
    coerce=False,
```

(continues on next page)

(continued from previous page)

```
    pandas_dtype=None,
    index=None,
    strict=False
    name=None,
    ordered=False
)>
```

This reclassifies an index (or indices) as a column (or columns).

Similarly, to partially alter the index, pass the name of the column you would like to be removed to the `level` parameter, and you may also decide whether to drop the levels with the `drop` parameter.

```
>>> example_schema = pa.DataFrameSchema({
...     "category" : pa.Column(pa.String)},
...     index = pa.MultiIndex([
...         pa.Index(name = "unique_id1", pandas_dtype = pa.Int),
...         pa.Index(name = "unique_id2", pandas_dtype = pa.String)
...         ]
...     )
... )
>>> print(example_schema.reset_index(level = ["unique_id1"]))
<Schema DataFrameSchema(
    columns={
        'category': <Schema Column(name=category, type=str)>
        'unique_id1': <Schema Column(name=unique_id1, type=int64)>
    },
    checks=[],
    coerce=False,
    pandas_dtype=None,
    index=<Schema Index(name=unique_id2, type=str)>,
    strict=False
    name=None,
    ordered=False
)>
```

See also:

set_index()

6.11.1.1.10 pandera.schemas.DataFrameSchema.select_columns

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

Select subset of columns in the schema.

New in version 0.4.5

Parameters **columns** (List[str]) – list of column names to select.

Return type *DataFrameSchema*

Returns *DataFrameSchema* (copy of original) with only the selected columns.

Raises *SchemaInitError* if column not in the schema.

Example

To subset a schema by column, and return a new schema:

```
>>> import pandera as pa
>>>
>>> example_schema = pa.DataFrameSchema({
...     "category" : pa.Column(pa.String),
...     "probability": pa.Column(pa.Float)
... })
>>>
>>> print(example_schema.select_columns(['category']))
<Schema DataFrameSchema(
    columns={
        'category': <Schema Column(name=category, type=str)>
    },
    checks=[],
    coerce=False,
    pandas_dtype=None,
    index=None,
    strict=False
    name=None,
    ordered=False
)>
```

Note: If an index is present in the schema, it will also be included in the new schema.

6.11.1.1.11 pandera.schemas.DataFrameSchema.set_index

DataFrameSchema.**set_index**(*keys*, *drop=True*, *append=False*)

A method for setting the Index of a *DataFrameSchema*, via an existing Column or list of columns.

Parameters

- **keys** (List[str]) – list of labels
- **drop** (bool) – bool, default True
- **append** (bool) – bool, default False

Return type *DataFrameSchema*

Returns a new *DataFrameSchema* with specified column(s) in the index.

Raises *SchemaInitError* if column not in the schema.

Examples

Just as you would set the index in a pandas DataFrame from an existing column, you can set an index within the schema from an existing column in the schema.

```
>>> import pandera as pa
>>>
>>> example_schema = pa.DataFrameSchema({
...     "category" : pa.Column(pa.String),
...     "probability": pa.Column(pa.Float)})
>>>
>>> print(example_schema.set_index(['category']))
<Schema DataFrameSchema(
    columns={
        'probability': <Schema Column(name=probability, type=float)>
```

(continues on next page)

(continued from previous page)

```
    },
    checks=[],
    coerce=False,
    pandas_dtype=None,
    index=<Schema Index(name=category, type=str)>,
    strict=False
    name=None,
    ordered=False
)>
```

If you have an existing index in your schema, and you would like to append a new column as an index to it (yielding a `Multiindex`), just use set_index as you would in pandas.

```
>>> example_schema = pa.DataFrameSchema(
...     {
...         "column1": pa.Column(pa.String),
...         "column2": pa.Column(pa.Int)
...     },
...     index=pa.Index(name = "column3", pandas_dtype = pa.Int)
... )
>>>
>>> print(example_schema.set_index(["column2"], append = True))
<Schema DataFrameSchema(
    columns={
        'column1': <Schema Column(name=column1, type=str)>
    },
    checks=[],
    coerce=False,
    pandas_dtype=None,
    index=<Schema MultiIndex(
        indexes=[
            <Schema Index(name=column3, type=int)>
            <Schema Index(name=column2, type=int)>
        ]
        coerce=False,
        strict=False,
        name=None,
        ordered=True
    )>,
    strict=False
    name=None,
    ordered=False
)>
```

See also:

reset_index()

6.11.1.1.12 pandera.schemas.DataFrameSchema.strategy

DataFrameSchema.**strategy**(*, *size=None*, *n_regex_columns=1*)
Create a `hypothesis` strategy for generating a DataFrame.

Parameters

- **size** – number of elements to generate
- **n_regex_columns** – number of regex columns to generate.

Returns a strategy that generates pandas DataFrame objects.

6.11.1.1.13 pandera.schemas.DataFrameSchema.to_script

DataFrameSchema.**to_script**(*fp=None*)
Create DataFrameSchema from yaml file.

Parameters **path** – str, Path to write script

Return type *DataFrameSchema*

Returns dataframe schema.

6.11.1.1.14 pandera.schemas.DataFrameSchema.to_yaml

DataFrameSchema.**to_yaml**(*stream=None*)
Write DataFrameSchema to yaml file.

Parameters **stream** (Optional[PathLike]) – file stream to write to. If None, dumps to string.

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

6.11.1.1.15 pandera.schemas.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

Raises *SchemaInitError*: if column not in schema or you try to change the name.

Example

Calling `schema.1` returns the *DataFrameSchema* with the updated column.

```
>>> import pandera as pa
>>>
>>> example_schema = pa.DataFrameSchema({
...     "category" : pa.Column(pa.String),
```

(continues on next page)

(continued from previous page)

```
...     "probability": pa.Column(pa.Float)
... })
>>> print(
...     example_schema.update_column(
...         'category', pandas_dtype=pa.Category
...     )
... )
<Schema DataFrameSchema(
    columns={
        'category': <Schema Column(name=category, type=category)>
        'probability': <Schema Column(name=probability, type=float)>
    },
    checks=[],
    coerce=False,
    pandas_dtype=None,
    index=None,
    strict=False
    name=None,
    ordered=False
)>
```

See also:

rename_columns()

6.11.1.1.16 pandera.schemas.DataFrameSchema.update_columns

DataFrameSchema.**update_columns**(*update_dict*)

Create copy of a *DataFrameSchema* with updated column properties.

Parameters **update_dict** (Dict[str, Dict[str, Any]]) –

Return type *DataFrameSchema*

Returns a new *DataFrameSchema* with updated columns

Raises *SchemaInitError*: if column not in schema or you try to change the name.

Example

Calling `schema.update_columns` returns the *DataFrameSchema* with the updated columns.

```
>>> import pandera as pa
>>>
>>> example_schema = pa.DataFrameSchema({
...     "category" : pa.Column(pa.String),
...     "probability": pa.Column(pa.Float)
... })
>>>
>>> print(
...     example_schema.update_columns(
...         {"category": {"pandas_dtype":pa.Category}}
...     )
... )
<Schema DataFrameSchema(
    columns={
        'category': <Schema Column(name=category, type=category)>
        'probability': <Schema Column(name=probability, type=float)>
```

(continues on next page)

(continued from previous page)

```
    },
    checks=[],
    coerce=False,
    pandas_dtype=None,
    index=None,
    strict=False
    name=None,
    ordered=False
)>
```

Note: This is the successor to the `update_column` method, which will be deprecated.

6.11.1.1.17 pandera.schemas.DataFrameSchema.validate

`DataFrameSchema.`**`validate`**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*, *inplace=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 `SchemaErrors`. Otherwise, raise `SchemaError` as soon as one occurs.
- **inplace** (bool) – if True, applies coercion to the object of validation, otherwise creates a copy of the data.

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"]
... })
>>>
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

6.11.1.1.18 pandera.schemas.DataFrameSchema.__call__

DataFrameSchema.**__call__**(*dataframe*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*, *inplace=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 SchemaErrors. Otherwise, raise SchemaError as soon as one occurs.
- **inplace** (bool) – if True, applies coercion to the object of validation, otherwise creates a copy of the data.

6.11.1.2 pandera.schemas.SeriesSchema

class pandera.schemas.**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, dtype, 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
pdtype	PandasDtype of the series.

Methods

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

6.11.1.2.1 pandera.schemas.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, dtype, 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*) –

6.11.1.2.2 pandera.schemas.SeriesSchema.validate

`SeriesSchema.`**validate**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*, *inplace=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 `SchemaErrors`. Otherwise, raise `SchemaError` as soon as one occurs.
- **inplace** (`bool`) – if True, applies coercion to the object of validation, otherwise creates a copy of the data.

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(
...     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
```

(continues on next page)

(continued from previous page)

```
4    999.0
dtype: float64
```

6.11.1.2.3 pandera.schemas.SeriesSchema.__call__

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

Alias for *SeriesSchema.validate()* method.

Return type Series

6.11.2 Schema Components

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

6.11.2.1 pandera.schema_components.Column

class pandera.schema_components.**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, dtype, 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 column can contain duplicate values.
- **coerce** (bool) – If True, when schema.validate is called the column will be coerced into the specified dtype. This has no effect on columns where pandas_dtype=None.
- **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
`pdtype`	PandasDtype of the series.
`properties`	Get column properties.
`regex`	True if `name` attribute should be treated as a regex pattern.

Methods

`__init__`	Create column validator object.
`coerce_dtype`	Coerce dtype of a column, handling duplicate column names.
`example`	Generate an example of a particular size.
`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.
`strategy`	Create a `hypothesis` strategy for generating a Column.
`strategy_component`	Generate column data object for use by DataFrame strategy.
`validate`	Validate a Column in a DataFrame object.
`__call__`	Alias for `validate` method.

6.11.2.1.1 pandera.schema_components.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, dtype, 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 column can contain duplicate values.
- **coerce** (bool) – If True, when schema.validate is called the column will be coerced into the specified dtype. This has no effect on columns where pandas_dtype=None.
- **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.

6.11.2.1.2 pandera.schema_components.Column.coerce_dtype

Column.**coerce_dtype**(*obj*)

Coerce dtype of a column, handling duplicate column names.

6.11.2.1.3 pandera.schema_components.Column.example

Column.**example**(*size=None*)

Generate an example of a particular size.

Parameters **size** – number of elements in the generated Index.

Return type DataFrame

Returns pandas DataFrame object.

6.11.2.1.4 pandera.schema_components.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

6.11.2.1.5 pandera.schema_components.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

6.11.2.1.6 pandera.schema_components.Column.strategy

Column.**strategy**(*, *size=None*)

Create a hypothesis strategy for generating a Column.

Parameters **size** – number of elements to generate

Returns a dataframe strategy for a single column.

6.11.2.1.7 pandera.schema_components.Column.strategy_component

Column.**strategy_component**()

Generate column data object for use by DataFrame strategy.

6.11.2.1.8 pandera.schema_components.Column.validate

Column.**validate**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*, *inplace=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 `SchemaErrors`. Otherwise, raise `SchemaError` as soon as one occurs.
- **inplace** (bool) – if True, applies coercion to the object of validation, otherwise creates a copy of the data.

Return type DataFrame

Returns validated DataFrame.

6.11.2.1.9 pandera.schema_components.Column.__call__

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

Alias for `validate` method.

Return type Union[DataFrame, Series]

6.11.2.2 pandera.schema_components.Index

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

Validate types and properties of a DataFrame Index.

Initialize series schema base object.

Parameters

- **pandas_dtype** (Union[str, type, *PandasDtype*, ExtensionDtype, dtype, 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.
- **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.
names	Get index names in the Index schema component.

continues on next page

Table 9 – continued from previous page

nullable	Whether the series is nullable.
pandas_dtype	Get the pandas dtype
pdtype	PandasDtype of the series.

Methods

example	Generate an example of a particular size.
strategy	Create a hypothesis strategy for generating an Index.
strategy_component	Generate column data object for use by MultiIndex strategy.
validate	Validate DataFrameSchema or SeriesSchema Index.
__call__	Alias for validate method.

6.11.2.2.1 pandera.schema_components.Index.example

Index.**example**(*size=None*)

Generate an example of a particular size.

Parameters **size** (Optional[int]) – number of elements in the generated Index.

Return type Index

Returns pandas Index object.

6.11.2.2.2 pandera.schema_components.Index.strategy

Index.**strategy**(*, *size=None*)

Create a hypothesis strategy for generating an Index.

Parameters **size** (Optional[int]) – number of elements to generate.

Returns index strategy.

6.11.2.2.3 pandera.schema_components.Index.strategy_component

Index.**strategy_component**()

Generate column data object for use by MultiIndex strategy.

6.11.2.2.4 pandera.schema_components.Index.validate

Index.**validate**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*, *inplace=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.
- **lazy** (bool) – if True, lazily evaluates dataframe against all validation checks and raises a `SchemaErrors`. Otherwise, raise `SchemaError` as soon as one occurs.
- **inplace** (bool) – if True, applies coercion to the object of validation, otherwise creates a copy of the data.

Return type Union[DataFrame, Series]

Returns validated DataFrame or Series.

6.11.2.2.5 pandera.schema_components.Index.__call__

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

Alias for `validate` method.

Return type Union[DataFrame, Series]

6.11.2.3 pandera.schema_components.MultiIndex

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

Validate types and properties of a DataFrame MultiIndex.

This class inherits from *DataFrameSchema* to leverage its validation logic.

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
- **ordered** (bool) – whether or not to validate the indexes order.

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"])),
```

(continues on next page)

(continued from previous page)

```
...                 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 or not to coerce data types.
`dtype`	A pandas style dtype dict where the keys are column names and values are pandas dtype for the column.
`names`	Get index names in the MultiIndex schema component.
`ordered`	Whether or not to validate the columns order.
`pandas_dtype`	Get the pandas dtype property.
`pdtype`	PandasDtype of the dataframe.

Methods

`__init__`	Create MultiIndex validator.
`coerce_dtype`	Coerce type of a pd.Series by type specified in pandas_dtype.
`example`	Generate an example of a particular size.
`strategy`	Create a `hypothesis` strategy for generating a DataFrame.
`validate`	Validate DataFrame or Series MultiIndex.
`__call__`	Alias for `DataFrameSchema.validate()` method.

6.11.2.3.1 pandera.schema_components.MultiIndex.__init__

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

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
- **ordered** (`bool`) – whether or not to validate the indexes order.

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.

6.11.2.3.2 pandera.schema_components.MultiIndex.coerce_dtype

MultiIndex.**coerce_dtype**(*obj*)

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

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

Return type MultiIndex

Returns MultiIndex with coerced data type

6.11.2.3.3 pandera.schema_components.MultiIndex.example

MultiIndex.**example**(*size=None*)

Generate an example of a particular size.

Parameters **size** – number of elements in the generated DataFrame.

Return type MultiIndex

Returns pandas DataFrame object.

6.11.2.3.4 pandera.schema_components.MultiIndex.strategy

MultiIndex.**strategy**(*, *size=None*)

Create a hypothesis strategy for generating a DataFrame.

Parameters

- **size** – number of elements to generate
- **n_regex_columns** – number of regex columns to generate.

Returns a strategy that generates pandas DataFrame objects.

6.11.2.3.5 pandera.schema_components.MultiIndex.validate

MultiIndex.**validate**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*, *inplace=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 SchemaErrors. Otherwise, raise SchemaError as soon as one occurs.

- **inplace** (bool) – if True, applies coercion to the object of validation, otherwise creates a copy of the data.

Return type Union[DataFrame, Series]

Returns validated DataFrame or Series.

6.11.2.3.6 pandera.schema_components.MultiIndex.__call__

MultiIndex.**__call__**(*dataframe*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*, *inplace=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 SchemaErrors. Otherwise, raise SchemaError as soon as one occurs.
- **inplace** (bool) – if True, applies coercion to the object of validation, otherwise creates a copy of the data.

6.11.3 Schema Models

pandera.model.SchemaModel	Definition of a DataFrameSchema.

6.11.3.1 pandera.model.SchemaModel

class pandera.model.**SchemaModel**(**args*, ***kwargs*)

Definition of a DataFrameSchema.

new in 0.5.0

See the *User Guide* for more.

Methods

example	Create a `hypothesis` strategy for generating a DataFrame.
strategy	Create a `hypothesis` strategy for generating a DataFrame.
to_schema	Create `DataFrameSchema` from the *SchemaModel*.
to_yaml	Convert *Schema* to yaml using *io.to_yaml*.
validate	Check if all columns in a dataframe have a column in the Schema.

6.11.3.1.1 pandera.model.SchemaModel.example

classmethod `SchemaModel.`**example**(*, *size=None*)

Create a `hypothesis` strategy for generating a DataFrame.

Parameters

- **size** – number of elements to generate
- **n_regex_columns** – number of regex columns to generate.

Returns a strategy that generates pandas DataFrame objects.

6.11.3.1.2 pandera.model.SchemaModel.strategy

classmethod `SchemaModel.`**strategy**(*, *size=None*)

Create a `hypothesis` strategy for generating a DataFrame.

Parameters

- **size** – number of elements to generate
- **n_regex_columns** – number of regex columns to generate.

Returns a strategy that generates pandas DataFrame objects.

6.11.3.1.3 pandera.model.SchemaModel.to_schema

classmethod `SchemaModel.`**to_schema**()

Create `DataFrameSchema` from the *SchemaModel*.

Return type *DataFrameSchema*

6.11.3.1.4 pandera.model.SchemaModel.to_yaml

classmethod SchemaModel.**to_yaml**(*stream=None*)

Convert *Schema* to yaml using *io.to_yaml*.

6.11.3.1.5 pandera.model.SchemaModel.validate

classmethod SchemaModel.**validate**(*check_obj*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*, *inplace=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 SchemaErrors. Otherwise, raise SchemaError as soon as one occurs.
- **inplace** (bool) – if True, applies coercion to the object of validation, otherwise creates a copy of the data.

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),
```

(continues on next page)

(continued from previous page)

```
...         ]),
... })
>>>
>>> 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
```

Model Components

pandera.model_components.Field	Used to provide extra information about a field of a SchemaModel.
pandera.model_components.check	Decorator to make SchemaModel method a column/index check function.
pandera.model_components.dataframe_check	Decorator to make SchemaModel method a dataframe-wide check function.

6.11.3.2 pandera.model_components.Field

pandera.model_components.**Field**(*, *eq=None*, *ne=None*, *gt=None*, *ge=None*, *lt=None*, *le=None*, *in_range=None*, *isin=None*, *notin=None*, *str_contains=None*, *str_endswith=None*, *str_length=None*, *str_matches=None*, *str_startswith=None*, *nullable=False*, *allow_duplicates=True*, *coerce=False*, *regex=False*, *ignore_na=True*, *raise_warning=False*, *n_failure_cases=10*, *alias=None*, *check_name=None*, *dtype_kwargs=None*, ***kwargs*)

Used to provide extra information about a field of a SchemaModel.

new in 0.5.0

Some arguments apply only to numeric dtypes and some apply only to `str`. See the *User Guide* for more information.

The keyword-only arguments from `eq` to `str_startswith` are dispatched to the built-in *~pandera.checks.Check* methods.

Parameters

- **nullable** (`bool`) – whether or not the column/index is nullable.
- **allow_duplicates** (`bool`) – whether or not to accept duplicate values.
- **coerce** (`bool`) – coerces the data type if `True`.
- **regex** (`bool`) – whether or not the field name or alias is a regex pattern.
- **ignore_na** (`bool`) – whether or not to ignore null values in the checks.
- **raise_warning** (`bool`) – raise a warning instead of an Exception.
- **n_failure_cases** (`int`) – report the first n unique failure cases. If None, report all failure cases.

- **alias** (Optional[Any]) – The public name of the column/index.
- **check_name** (Optional[bool]) – Whether to check the name of the column/index during validation. *None* is the default behavior, which translates to *True* for columns and multi-index, and to *False* for a single index.
- **dtype_kwargs** (Optional[Dict[str, Any]]) – The parameters to be forwarded to the type of the field.
- **kwargs** – Specify custom checks that have been registered with the *register_check_method* decorator.

Return type Any

6.11.3.3 pandera.model_components.check

pandera.model_components.**check**(**fields*, *regex=False*, ***check_kwargs*)

Decorator to make SchemaModel method a column/index check function.

new in 0.5.0

This indicates that the decorated method should be used to validate a field (column or index). The method will be converted to a classmethod. Therefore its signature must start with *cls* followed by regular check arguments. See the *User Guide* for more.

Parameters

- **_fn** – Method to decorate.
- **check_kwargs** – Keywords arguments forwarded to Check.

Return type Callable[[Union[classmethod, Callable[..., Any]]], classmethod]

6.11.3.4 pandera.model_components.dataframe_check

pandera.model_components.**dataframe_check**(*_fn=None*, ***check_kwargs*)

Decorator to make SchemaModel method a dataframe-wide check function.

new in 0.5.0

Decorate a method on the SchemaModel indicating that it should be used to validate the DataFrame. The method will be converted to a classmethod. Therefore its signature must start with *cls* followed by regular check arguments. See the *User Guide* for more.

Parameters **check_kwargs** – Keywords arguments forwarded to Check.

Return type Callable[[Union[classmethod, Callable[..., Any]]], classmethod]

Typing

pandera.typing	Typing definitions and helpers.

6.11.3.5 pandera.typing

Typing definitions and helpers.

Pandas object annotations

`DataFrame`([data, index, columns, dtype, copy])	Representation of pandas.DataFrame, only used for type annotation.
`Index`([data, dtype, copy, name, tupleize_cols])	Representation of pandas.Index, only used for type annotation.
`Series`([data, index, dtype, name, copy, ...])	Representation of pandas.Series, only used for type annotation.

Dtype annotations

`Bool`	`"bool"` numpy dtype
`DateTime`	`"datetime64[ns]"` numpy dtype
`Timedelta`	`"timedelta64[ns]"` numpy dtype
`Category`	pandas `"categorical"` datatype
`Float`	`"float"` numpy dtype
`Float16`	`"float16"` numpy dtype
`Float32`	`"float32"` numpy dtype
`Float64`	`"float64"` numpy dtype
`Int`	`"int"` numpy dtype
`Int8`	`"int8"` numpy dtype
`Int16`	`"int16"` numpy dtype
`Int32`	`"int32"` numpy dtype
`Int64`	`"int64"` numpy dtype
`UInt8`	`"uint8"` numpy dtype
`UInt16`	`"uint16"` numpy dtype
`UInt32`	`"uint32"` numpy dtype
`UInt64`	`"uint64"` numpy dtype
`INT8`	`"Int8"` pandas dtype:: pandas 0.24.0+
`INT16`	`"Int16"` pandas dtype: pandas 0.24.0+
`INT32`	`"Int32"` pandas dtype: pandas 0.24.0+
`INT64`	`"Int64"` pandas dtype: pandas 0.24.0+
`UINT8`	`"UInt8"` pandas dtype:: pandas 0.24.0+
`UINT16`	`"UInt16"` pandas dtype: pandas 0.24.0+
`UINT32`	`"UInt32"` pandas dtype: pandas 0.24.0+
`UINT64`	`"UInt64"` pandas dtype: pandas 0.24.0+
`Object`	`"object"` numpy dtype
`String`	`"str"` numpy dtype
`STRING`	`"str"` numpy dtype

Config

`pandera.model.BaseConfig`	Define DataFrameSchema-wide options.

6.11.3.6 pandera.model.BaseConfig

class pandera.model.**BaseConfig**
Bases: object

Define DataFrameSchema-wide options.

new in 0.5.0

Attributes

coerce	coerce types of all schema components
multiindex_coerce	coerce types of all MultiIndex components
multiindex_name	name of multiindex
multiindex_ordered	validate MultiIndex in order
multiindex_strict	make sure all specified columns are in validated MultiIndex - if "filter", removes indexes not specified in the schema
name	name of schema
ordered	validate columns order
strict	make sure all specified columns are in the validated dataframe - if "filter", removes columns not specified in the schema

6.11.4 Checks

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

6.11.4.1 pandera.checks.Check

class pandera.checks.**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, null values will be ignored when determining if a check passed or failed. For dataframes, ignores 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 first n unique failure cases. If None, 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"],
... )
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # 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],
...     "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.

continues on next page

Table 23 – continued from previous page

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.

6.11.4.1.1 pandera.checks.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

6.11.4.1.2 pandera.checks.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

6.11.4.1.3 pandera.checks.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

6.11.4.1.4 pandera.checks.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

6.11.4.1.5 pandera.checks.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

6.11.4.1.6 pandera.checks.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

6.11.4.1.7 pandera.checks.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

6.11.4.1.8 pandera.checks.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()`.

6.11.4.1.9 pandera.checks.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

6.11.4.1.10 pandera.checks.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

6.11.4.1.11 pandera.checks.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

6.11.4.1.12 pandera.checks.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

6.11.4.1.13 pandera.checks.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

6.11.4.1.14 pandera.checks.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

6.11.4.1.15 pandera.checks.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.

6.11.4.1.16 pandera.checks.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()`.

6.11.4.1.17 pandera.checks.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

6.11.4.1.18 pandera.checks.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

6.11.4.1.19 pandera.checks.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().

6.11.4.1.20 pandera.checks.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

6.11.4.1.21 pandera.checks.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.

6.11.4.2 pandera.hypotheses.Hypothesis

class pandera.hypotheses.**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* as-

sesses 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.

6.11.4.2.1 pandera.hypotheses.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* as-

sesses 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.

6.11.4.2.2 pandera.hypotheses.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
```

6.11.4.2.3 pandera.hypotheses.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
```

6.11.4.2.4 pandera.hypotheses.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.

6.11.5 Pandas Data Types

pandera.dtypes.PandasDtype	Enumerate all valid pandas data types.

6.11.5.1 pandera.dtypes.PandasDtype

class pandera.dtypes.**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
`Complex`	`"complex"` numpy dtype
`Complex128`	`"complex"` numpy dtype
`Complex256`	`"complex"` numpy dtype
`Complex64`	`"complex"` numpy dtype
`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`	`"string"` pandas dtypes: pandas 1.0.0+.
`String`	`"str"` numpy dtype
`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

6.11.6 Decorators

pandera.decorators.check_input	Validate function argument when function is called.
pandera.decorators.check_output	Validate function output.
pandera.decorators.check_io	Check schema for multiple inputs and outputs.
pandera.decorators.check_types	Validate function inputs and output based on type annotations.

6.11.6.1 pandera.decorators.check_input

pandera.decorators.**check_input**(*schema*, *obj_getter=None*, *head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*, *inplace=False*)
Validate function argument when function is called.

This is a decorator function that validates the schema of a dataframe argument in a 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 SchemaErrors. Otherwise, raise SchemaError as soon as one occurs.
- **inplace** (bool) – if True, applies coercion to the object of validation, otherwise creates a copy of the data.

Return type Callable[[~F], ~F]

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.

6.11.6.2 pandera.decorators.check_output

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

Validate function output.

Similar to input validator, but validates the output of the decorated function.

Parameters

- **schema** (Union[*DataFrameSchema*, *SeriesSchema*]) – dataframe/series schema object
- **obj_getter** (Union[str, int, 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 `SchemaErrors`. Otherwise, raise `SchemaError` as soon as one occurs.
- **inplace** (bool) – if True, applies coercion to the object of validation, otherwise creates a copy of the data.

Return type Callable[[~F], ~F]

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.

6.11.6.3 pandera.decorators.check_io

pandera.decorators.**check_io**(*head=None*, *tail=None*, *sample=None*, *random_state=None*, *lazy=False*, *inplace=False*, *out=None*, ***inputs*)

Check schema for multiple inputs and outputs.

See *here* for more usage details.

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.
- **lazy** (bool) – if True, lazily evaluates dataframe against all validation checks and raises a `SchemaErrors`. Otherwise, raise `SchemaError` as soon as one occurs.
- **inplace** (bool) – if True, applies coercion to the object of validation, otherwise creates a copy of the data.
- **out** (Union[*DataFrameSchema*, *SeriesSchema*, Tuple[Union[str, int, Callable], Union[*DataFrameSchema*, *SeriesSchema*]], List[Tuple[Union[str, int, Callable], Union[*DataFrameSchema*, *SeriesSchema*]]], None]) – this should be a schema object if the function outputs a single dataframe/series. It can be a two-tuple, where the first element is a string, integer, or callable that fetches the pandas data structure in the output, and the second element is the schema to validate against. For multiple outputs, specify a list of two-tuples following the above structure.
- **inputs** (Union[*DataFrameSchema*, *SeriesSchema*]) – kwargs keys should be the argument name in the decorated function and values should be the schema used to validate the pandas data structure referenced by the argument name.

Return type Callable[[~F], ~F]

Returns wrapped function

6.11.6.4 pandera.decorators.check_types

pandera.decorators.**check_types**(*wrapped: F*, ***, *head: Optional[int] = 'None'*, *tail: Optional[int] = 'None'*, *sample: Optional[int] = 'None'*, *random_state: Optional[int] = 'None'*, *lazy: bool = 'False'*, *inplace: bool = 'False'*) → F

pandera.decorators.**check_types**(*wrapped: None = None*, ***, *head: Optional[int] = 'None'*, *tail: Optional[int] = 'None'*, *sample: Optional[int] = 'None'*, *random_state: Optional[int] = 'None'*, *lazy: bool = 'False'*, *inplace: bool = 'False'*) → Callable[[F], F]

Validate function inputs and output based on type annotations.

See the *User Guide* for more.

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.
- **lazy** (bool) – if True, lazily evaluates dataframe against all validation checks and raises a `SchemaErrors`. Otherwise, raise `SchemaError` as soon as one occurs.
- **inplace** (bool) – if True, applies coercion to the object of validation, otherwise creates a copy of the data.

Return type Callable

6.11.7 Schema Inference

pandera.schema_inference.infer_schema	Infer schema for pandas DataFrame or Series object.

6.11.7.1 pandera.schema_inference.infer_schema

pandera.schema_inference.**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.

6.11.8 IO Utils

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

6.11.8.1 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.

6.11.8.2 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.

6.11.8.3 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.

6.11.9 Data Synthesis Strategies

pandera.strategies	Generate synthetic data from a schema definition.

6.11.9.1 pandera.strategies

Generate synthetic data from a schema definition.

new in 0.6.0

This module is responsible for generating data based on the type and check constraints specified in a pandera schema. It's built on top of the hypothesis package to compose strategies given multiple checks specified in a schema.

See the *user guide* for more details.

pandera.strategies.**column_strategy**(*pandas_dtype*, *strategy=None*, ***, *checks=None*, *allow_duplicates=True*, *name=None*)

Create a data object describing a column in a DataFrame.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **checks** (Optional[Sequence]) – sequence of *Check* s to constrain the values of the data in the column/index.
- **allow_duplicates** (Optional[bool]) – whether or not generated Series contains duplicates.
- **name** (Optional[str]) – name of the Series.

Returns a column object.

pandera.strategies.**dataframe_strategy**(*pandas_dtype=None*, *strategy=None*, ***, *columns=None*, *checks=None*, *index=None*, *size=None*, *n_regex_columns=1*)

Strategy to generate a pandas DataFrame.

Parameters

- **pandas_dtype** (Optional[*PandasDtype*]) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – if specified, this will raise a BaseStrategyOnlyError, since it cannot be chained to a prior strategy.
- **columns** (Optional[Dict]) – a dictionary where keys are column names and values are *Column* objects.
- **checks** (Optional[Sequence]) – sequence of *Check* s to constrain the values of the data at the dataframe level.
- **index** (Optional[Any]) – Index or MultiIndex schema component.
- **size** (Optional[int]) – number of elements in the Series.
- **n_regex_columns** (int) – number of regex columns to generate.

Returns hypothesis strategy.

pandera.strategies.**eq_strategy**(*pandas_dtype*, *strategy=None*, ***, *value*)

Strategy to generate a single value.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **value** (Any) – value to generate.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**field_element_strategy**(*pandas_dtype*, *strategy=None*, ***, *checks=None*)

Strategy to generate elements of a column or index.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **checks** (Optional[Sequence]) – sequence of *Check* s to constrain the values of the data in the column/index.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**ge_strategy**(*pandas_dtype*, *strategy=None*, ***, *min_value*)

Strategy to generate values greater than or equal to a minimum value.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **min_value** (Union[int, float]) – generate values greater than or equal to this.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**gt_strategy**(*pandas_dtype*, *strategy=None*, ***, *min_value*)

Strategy to generate values greater than a minimum value.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **min_value** (Union[int, float]) – generate values larger than this.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**in_range_strategy**(*pandas_dtype*, *strategy=None*, ***, *min_value*, *max_value*, *include_min=True*, *include_max=True*)

Strategy to generate values within a particular range.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **min_value** (Union[int, float]) – generate values greater than this.
- **max_value** (Union[int, float]) – generate values less than this.
- **include_min** (bool) – include min_value in generated data.
- **include_max** (bool) – include max_value in generated data.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**index_strategy**(*pandas_dtype*, *strategy=None*, ***, *checks=None*, *nullable=False*, *allow_duplicates=True*, *name=None*, *size=None*)

Strategy to generate a pandas Index.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **checks** (Optional[Sequence]) – sequence of *Check* s to constrain the values of the data in the column/index.
- **nullable** (Optional[bool]) – whether or not generated Series contains null values.

- **allow_duplicates** (Optional[bool]) – whether or not generated Series contains duplicates.
- **name** (Optional[str]) – name of the Series.
- **size** (Optional[int]) – number of elements in the Series.

Returns hypothesis strategy.

pandera.strategies.**isin_strategy**(*pandas_dtype*, *strategy=None*, ***, *allowed_values*)

Strategy to generate values within a finite set.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **allowed_values** (Sequence[Any]) – set of allowable values.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**le_strategy**(*pandas_dtype*, *strategy=None*, ***, *max_value*)

Strategy to generate values less than or equal to a maximum value.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **max_value** (Union[int, float]) – generate values less than or equal to this.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**lt_strategy**(*pandas_dtype*, *strategy=None*, ***, *max_value*)

Strategy to generate values less than a maximum value.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **max_value** (Union[int, float]) – generate values less than this.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**multiindex_strategy**(*pandas_dtype=None*, *strategy=None*, ***, *indexes=None*, *size=None*)

Strategy to generate a pandas MultiIndex object.

Parameters

- **pandas_dtype** (Optional[*PandasDtype*]) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.

- **indexes** (Optional[List]) – a list of Inded objects.
- **size** (Optional[int]) – number of elements in the Series.

Returns hypothesis strategy.

pandera.strategies.**ne_strategy**(*pandas_dtype*, *strategy=None*, ***, *value*)
Strategy to generate anything except for a particular value.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **value** (Any) – value to avoid.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**notin_strategy**(*pandas_dtype*, *strategy=None*, ***, *forbidden_values*)
Strategy to generate values excluding a set of forbidden values

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **forbidden_values** (Sequence[Any]) – set of forbidden values.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**numpy_complex_dtypes**(*dtype*, *min_value=0j*, *max_value=None*, *allow_infinity=None*, *allow_nan=None*)
Create numpy strategy for complex numbers.

Parameters

- **dtype** – numpy complex number datatype
- **min_value** (complex) – minimum value, must be complex number
- **max_value** (Optional[complex]) – maximum value, must be complex number

Returns hypothesis strategy

pandera.strategies.**numpy_time_dtypes**(*dtype*, *min_value=None*, *max_value=None*)
Create numpy strategy for datetime and timedelta data types.

Parameters

- **dtype** – numpy datetime or timedelta datatype
- **min_value** – minimum value of the datatype to create
- **max_value** – maximum value of the datatype to create

Returns hypothesis strategy

pandera.strategies.**pandas_dtype_strategy**(*pandas_dtype*, *strategy=None*, ***kwargs*)
Strategy to generate data from a *pandera.dtypes.PandasDtype*.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.

Kwargs key-word arguments passed into hypothesis.extra.numpy.from_dtype . For datetime, timedelta, and complex number datatypes, these arguments are passed into *numpy_time_dtypes()* and *numpy_complex_dtypes()*.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**register_check_strategy**(*strategy_fn*)

Decorate a Check method with a strategy.

This should be applied to a built-in *Check* method.

Parameters **strategy_fn** (Callable[..., SearchStrategy]) – add strategy to a check, using check statistics to generate a hypothesis strategy.

pandera.strategies.**series_strategy**(*pandas_dtype*, *strategy=None*, ***, *checks=None*, *nullable=False*, *allow_duplicates=True*, *name=None*, *size=None*)

Strategy to generate a pandas Series.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **checks** (Optional[Sequence]) – sequence of *Check* s to constrain the values of the data in the column/index.
- **nullable** (Optional[bool]) – whether or not generated Series contains null values.
- **allow_duplicates** (Optional[bool]) – whether or not generated Series contains duplicates.
- **name** (Optional[str]) – name of the Series.
- **size** (Optional[int]) – number of elements in the Series.

Returns hypothesis strategy.

pandera.strategies.**str_contains_strategy**(*pandas_dtype*, *strategy=None*, ***, *pattern*)

Strategy to generate strings that contain a particular pattern.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **pattern** (str) – regex pattern.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**str_endswith_strategy**(*pandas_dtype*, *strategy=None*, ***, *string*)

Strategy to generate strings that end with a specific string pattern.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **string** (str) – string pattern.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**str_length_strategy**(*pandas_dtype*, *strategy=None*, ***, *min_value*, *max_value*)

Strategy to generate strings of a particular length

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **min_value** (int) – minimum string length.
- **max_value** (int) – maximum string length.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**str_matches_strategy**(*pandas_dtype*, *strategy=None*, ***, *pattern*)

Strategy to generate strings that patch a regex pattern.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **pattern** (str) – regex pattern.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**str_startswith_strategy**(*pandas_dtype*, *strategy=None*, ***, *string*)

Strategy to generate strings that start with a specific string pattern.

Parameters

- **pandas_dtype** (*PandasDtype*) – *pandera.dtypes.PandasDtype* instance.
- **strategy** (Optional[SearchStrategy]) – an optional hypothesis strategy. If specified, the pandas dtype strategy will be chained onto this strategy.
- **string** (str) – string pattern.

Return type SearchStrategy

Returns hypothesis strategy

pandera.strategies.**verify_pandas_dtype**(*pandas_dtype*, *schema_type*, *name*)

Verify that pandas_dtype argument is not None.

6.11.10 Extensions

pandera.extensions	pandera API extensions

6.11.10.1 pandera.extensions

pandera API extensions

new in 0.6.0

This module provides utilities for extending the `pandera` API.

class pandera.extensions.**CheckType**(*value*)

Bases: `enum.Enum`

Check types for registered check methods.

VECTORIZED = 1

Check applied to a Series or DataFrame

ELEMENT_WISE = 2

Check applied to an element of a Series or DataFrame

GROUPBY = 3

Check applied to dictionary of Series or DataFrames.

pandera.extensions.**register_check_method**(*check_fn=None*, ***, *statistics=None*, *supported_types=(<class 'pandas.core.frame.DataFrame'>, <class 'pandas.core.series.Series'>)*, *check_type='vectorized'*, *strategy=None*)

Registers a function as a *Check* method.

See the *user guide* for more details.

Parameters

- **check_fn** – check function to register. The function should take one positional argument for the object to validate and additional keyword-only arguments for the check statistics.
- **statistics** (Optional[List[str]]) – list of keyword-only arguments in the `check_fn`, which serve as the statistics needed to serialize/de-serialize the check and generate data if a `strategy` function is provided.
- **supported_types** (Union[type, Tuple, List]) – the pandas type(s) supported by the check function. Valid values are `pd.DataFrame`, `pd.Series`, or a list/tuple of `(pa.DataFrame, pa.Series)` if both types are supported.
- **check_type** (Union[*CheckType*, str]) – the expected input of the check function. Valid values are *CheckType* enums or `{"vectorized", "element_wise", "groupby"}`. The input signature of `check_fn` is determined by this argument:
  - if `vectorized`, the first positional argument of `check_fn` should be one of the `supported_types`.
  - if `element_wise`, the first positional argument of `check_fn` should be a single scalar element in the pandas Series or DataFrame.
  - if `groupby`, the first positional argument of `check_fn` should be a dictionary mapping group names to subsets of the Series or DataFrame.

- **strategy** – data-generation strategy associated with the check function.

Returns register check function wrapper.

6.11.11 Errors

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

6.11.11.1 pandera.errors.SchemaError

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

Raised when object does not pass schema validation constraints.

6.11.11.2 pandera.errors.SchemaErrors

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

Raised when multiple schema are lazily collected into one error.

6.11.11.3 pandera.errors.SchemaInitError

class pandera.errors.**SchemaInitError**

Raised when schema initialization fails.

6.11.11.4 pandera.errors.SchemaDefinitionError

class pandera.errors.**SchemaDefinitionError**

Raised when schema definition is invalid on object validation.

6.12 Contributing

Whether you are a novice or experienced software developer, all contributions and suggestions are welcome!

6.12.1 Getting Started

If you are looking to contribute to the *pandera* codebase, the best place to start is the GitHub "issues" tab. This is also a great place for filing bug reports and making suggestions for ways in which we can improve the code and documentation.

6.12.2 Contributing to the Codebase

The code is hosted on GitHub, so you will need to use Git to clone the project and make changes to the codebase. Once you have obtained a copy of the code, you should create a development environment that is separate from your existing Python environment so that you can make and test changes without compromising your own work environment.

An excellent guide on setting up python environments can be found here. Pandera offers a `environment.yml` to set up a conda-based environment and `requirements-dev.txt` for a virtualenv.

6.12.2.1 Project Releases

Releases are organized under milestones, which are be associated with a corresponding branch. This project uses semantic versioning, and we recommend prioritizing issues associated with the next release.

6.12.2.2 Contributing Documentation

Maybe the easiest, fastest, and most useful way to contribute to this project (and any other project) is to contribute documentation. If you find an API within the project that doesn't have an example or description, or could be clearer in its explanation, contribute yours!

You can also find issues for improving documentation under the docs label. If you have ideas for documentation improvements, you can create a new issue here

This project uses Sphinx for auto-documentation and RST syntax for docstrings. Once you have the code downloaded and you find something that is in need of some TLD, take a look at the Sphinx documentation or well-documented examples within the codebase for guidance on contributing.

You can build the html documentation by running `nox -s docs`. The built documentation can be found in `docs/_build`.

6.12.2.3 Contributing Bugfixes

Bugs are reported under the bug label, so if you find a bug create a new issue here.

6.12.2.4 Contributing Enhancements

New feature issues can be found under the enhancements label. You can request a feature by creating a new issue here.

6.12.2.5 Set up `pre-commit`

This project uses pre-commit to ensure that code standard checks pass locally before pushing to the remote project repo. Follow the installation instructions, then set up hooks with `pre-commit install`. After, `black`, `pylint` and `mypy` checks should be run with every commit.

6.12.2.6 Run the test suite locally

Before submitting your changes for review, make sure to check that your changes do not break any tests by running:

```
# if you're working with virtualenv
$ make nox

# if you're working with conda
$ make nox-conda
```

6.12.2.6.1 Using `mamba` (optional)

You can also use mamba, which is a faster implementation of miniconda, to run the `nox` test suite. Simply install it via conda-forge, and `make nox-conda` should use it under the hood.

```
$ conda install -c conda-forge mamba
$ make nox-conda
```

6.12.2.7 Making Pull Requests

Once your changes are ready to be submitted, make sure to push your changes to your fork of the GitHub repo before creating a pull request. Depending on the type of issue the pull request is resolving, your pull request should merge onto the appropriate branch:

6.12.2.7.1 Bugfixes

- branch naming convention: `bugfix/<issue number>` or `bugfix/<bugfix-name>`
- pull request to: `dev`

6.12.2.7.2 Documentation

- branch naming convention: `docs/<issue number>` or `docs/<doc-name>`
- pull request to: `release/x.x.x` branch if specified in the issue milestone, otherwise `dev`

6.12.2.7.3 Enhancements

- branch naming convention: `feature/<issue number>` or `feature/<bugfix-name>`
- pull request to: `release/x.x.x` branch if specified in the issue milestone, otherwise `dev`

We will review your changes, and might ask you to make additional changes before it is finally ready to merge. However, once it's ready, we will merge it, and you will have successfully contributed to the codebase!

6.12.2.8 Questions, Ideas, General Discussion

Head on over to the discussion section if you have questions or ideas, want to show off something that you did with `pandera`, or want to discuss a topic related to the project.

6.12.2.9 Dataframe Schema Style Guides

We have guidelines regarding dataframe and schema styles that are encouraged for each pull request:

- If specifying a single column DataFrame, this can be expressed as a one-liner:

```
DataFrameSchema({"col1": Column(...)})
```

- If specifying one column with multiple lines, or multiple columns:

```
DataFrameSchema(
    {
        "col1": Column(
            int,
            checks=[
                Check(...),
                Check(...),
            ]
        ),
    }
)
```

- If specifying columns with additional arguments that fit in one line:

```
DataFrameSchema(
    {"a": Column(int, nullable=True)},
    strict=True
)
```

- If specifying columns with additional arguments that don't fit in one line:

```
DataFrameSchema(
    {
        "a": Column(
            int,
            nullable=True,
            coerce=True,
            ...
        ),
        "b": Column(
            ...,
        )
```

(continues on next page)

(continued from previous page)

```
    },
    strict=True)
```

CHAPTER

SEVEN

HOW TO CITE

If you use `pandera` in the context of academic or industry research, please consider citing the paper and/or software package.

7.1 Paper

```
@InProceedings{ niels_bantilan-proc-scipy-2020,
  author    = { {N}iels {B}antilan },
  title     = { pandera: {S}tatistical {D}ata {V}alidation of {P}andas {D}ataframes },
  booktitle = { {P}roceedings of the 19th {P}ython in {S}cience {C}onference },
  pages     = { 116 - 124 },
  year      = { 2020 },
  editor    = { {M}eghann {A}garwal and {C}hris {C}alloway and {D}illon {N}iederhut␣
↪and {D}avid {S}hupe },
  doi       = { 10.25080/Majora-342d178e-010 }
}
```

7.2 Software Package

CHAPTER

EIGHT

LICENSE AND CREDITS

`pandera` is licensed under the MIT license. and is written and maintained by Niels Bantilan (niels@pandera.ci)

CHAPTER

NINE

INDICES AND TABLES

- genindex

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V