
pandera

Niels Bantilan, Nigel Markey, Jean-Francois Zinque

Dec 31, 2021

INTRODUCTION

1	Install	3
1.1	Extras	3
2	Quick Start	5
3	Schema Model	7
4	Informative Errors	9
5	Contributing	11
6	Issues	13
7	Need Help?	15
7.1	DataFrame Schemas	15
7.1.1	Column Validation	16
7.1.1.1	Null Values in Columns	16
7.1.1.2	Coercing Types on Columns	17
7.1.1.3	Required Columns	18
7.1.1.4	Ordered Columns	19
7.1.1.5	Stand-alone Column Validation	19
7.1.1.6	Column Regex Pattern Matching	19
7.1.1.7	Handling Dataframe Columns not in the Schema	21
7.1.1.8	Validating the order of the columns	22
7.1.1.9	Validating the joint uniqueness of columns	22
7.1.2	Index Validation	23
7.1.3	MultiIndex Validation	24
7.1.3.1	MultiIndex Columns	24
7.1.3.2	MultiIndex Indexes	24
7.1.4	Get Pandas Data Types	25
7.1.5	DataFrameSchema Transformations	26
7.2	Schema Models	28
7.2.1	Basic Usage	28
7.2.2	Validate on Initialization	29
7.2.3	Converting to DataFrameSchema	30
7.2.4	Excluded attributes	31
7.2.5	Supported dtypes	31
7.2.5.1	Dtype aliases	31
7.2.5.2	Type Vs instance	31
7.2.5.3	Parametrized dtypes	32
7.2.5.3.1	Annotated	32

7.2.5.3.2	Field	32
7.2.6	Required Columns	33
7.2.7	Schema Inheritance	33
7.2.8	Config	34
7.2.9	MultiIndex	34
7.2.10	Custom Checks	35
7.2.10.1	Column/Index checks	35
7.2.10.2	DataFrame Checks	37
7.2.10.3	Inheritance	37
7.2.11	Aliases	38
7.3	Series Schemas	39
7.4	Checks	40
7.4.1	Checking column properties	40
7.4.2	Built-in Checks	40
7.4.3	Vectorized vs. Element-wise Checks	41
7.4.4	Handling Null Values	41
7.4.5	Column Check Groups	41
7.4.6	Wide Checks	42
7.4.7	Raise UserWarning on Check Failure	43
7.4.8	Registering Custom Checks	45
7.5	Hypothesis Testing	45
7.5.1	Overview	45
7.5.2	Wide Hypotheses	46
7.6	Pandera Data Types	48
7.6.1	Motivations	48
7.6.2	DataType basics	48
7.6.3	Example	48
7.6.4	Parametrized data types	50
7.7	Decorators for Pipeline Integration	51
7.7.1	Check Input	51
7.7.2	Check Output	52
7.7.3	Check IO	53
7.7.4	Decorate Functions and Coroutines	54
7.8	Schema Inference	55
7.8.1	Schema Persistence	56
7.8.1.1	Write to a Python script	56
7.8.1.2	Write to YAML	57
7.9	Lazy Validation	58
7.10	Data Synthesis Strategies	61
7.10.1	Basic Usage	61
7.10.2	Usage in Unit Tests	61
7.10.3	Strategies and Examples from Schema Models	62
7.10.4	Checks as Constraints	63
7.10.4.1	Check Strategy Chaining	63
7.10.4.2	In-line Custom Checks	64
7.10.5	Defining Custom Strategies	64
7.11	Extensions	64
7.11.1	Registering Custom Check Methods	64
7.11.2	Specifying a Check Strategy	65
7.11.3	Check Types	67
7.11.3.1	Element-wise Checks	67
7.11.3.2	Groupby Checks	67
7.11.4	Registered Custom Checks with the Class-based API	68
7.12	Frictionless Data Schema	69

7.13	Supported DataFrame Libraries (New)	71
7.13.1	Scaling Up Data Validation	72
7.13.1.1	Data Validation with Dask	72
7.13.1.2	Data Validation with Fugue	74
7.13.1.2.1	What is Fugue?	74
7.13.1.2.2	Example	74
7.13.1.2.3	Validation by Partition	75
7.13.1.3	Data Validation with Koalas	77
7.13.1.4	Data Validation with Modin	78
7.14	Integrations	80
7.14.1	Pydantic	80
7.14.2	Mypy	81
7.14.2.1	Limitations	83
7.15	API	84
7.15.1	Core	84
7.15.1.1	Schemas	84
7.15.1.1.1	pandera.schemas.DataFrameSchema	84
7.15.1.1.1.1	pandera.schemas.DataFrameSchema.__init__	87
7.15.1.1.1.2	pandera.schemas.DataFrameSchema.add_columns	88
7.15.1.1.1.3	pandera.schemas.DataFrameSchema.coerce_dtype	89
7.15.1.1.1.4	pandera.schemas.DataFrameSchema.example	89
7.15.1.1.1.5	pandera.schemas.DataFrameSchema.from_yaml	89
7.15.1.1.1.6	pandera.schemas.DataFrameSchema.get_dtypes	90
7.15.1.1.1.7	pandera.schemas.DataFrameSchema.remove_columns	90
7.15.1.1.1.8	pandera.schemas.DataFrameSchema.rename_columns	91
7.15.1.1.1.9	pandera.schemas.DataFrameSchema.reset_index	92
7.15.1.1.1.10	pandera.schemas.DataFrameSchema.select_columns	93
7.15.1.1.1.11	pandera.schemas.DataFrameSchema.set_index	94
7.15.1.1.1.12	pandera.schemas.DataFrameSchema.strategy	95
7.15.1.1.1.13	pandera.schemas.DataFrameSchema.to_script	96
7.15.1.1.1.14	pandera.schemas.DataFrameSchema.to_yaml	96
7.15.1.1.1.15	pandera.schemas.DataFrameSchema.update_column	96
7.15.1.1.1.16	pandera.schemas.DataFrameSchema.update_columns	97
7.15.1.1.1.17	pandera.schemas.DataFrameSchema.validate	98
7.15.1.1.1.18	pandera.schemas.DataFrameSchema.__call__	99
7.15.1.1.2	pandera.schemas.SeriesSchema	99
7.15.1.1.2.1	pandera.schemas.SeriesSchema.__init__	101
7.15.1.1.2.2	pandera.schemas.SeriesSchema.validate	101
7.15.1.1.2.3	pandera.schemas.SeriesSchema.__call__	102
7.15.1.2	Schema Components	102
7.15.1.2.1	pandera.schema_components.Column	103
7.15.1.2.1.1	pandera.schema_components.Column.__init__	104
7.15.1.2.1.2	pandera.schema_components.Column.coerce_dtype	106
7.15.1.2.1.3	pandera.schema_components.Column.example	106
7.15.1.2.1.4	pandera.schema_components.Column.get_regex_columns	106
7.15.1.2.1.5	pandera.schema_components.Column.set_name	106
7.15.1.2.1.6	pandera.schema_components.Column.strategy	106
7.15.1.2.1.7	pandera.schema_components.Column.strategy_component	106
7.15.1.2.1.8	pandera.schema_components.Column.validate	107
7.15.1.2.1.9	pandera.schema_components.Column.__call__	107
7.15.1.2.2	pandera.schema_components.Index	107
7.15.1.2.2.1	pandera.schema_components.Index.example	109
7.15.1.2.2.2	pandera.schema_components.Index.strategy	109
7.15.1.2.2.3	pandera.schema_components.Index.strategy_component	109

7.15.1.2.2.4	pandera.schema_components.Index.validate	109
7.15.1.2.2.5	pandera.schema_components.Index.__call__	110
7.15.1.2.3	pandera.schema_components.MultiIndex	110
7.15.1.2.3.1	pandera.schema_components.MultiIndex.__init__	111
7.15.1.2.3.2	pandera.schema_components.MultiIndex.coerce_dtype	112
7.15.1.2.3.3	pandera.schema_components.MultiIndex.example	113
7.15.1.2.3.4	pandera.schema_components.MultiIndex.strategy	113
7.15.1.2.3.5	pandera.schema_components.MultiIndex.validate	113
7.15.1.2.3.6	pandera.schema_components.MultiIndex.__call__	114
7.15.1.3	Checks	114
7.15.1.3.1	pandera.checks.Check	114
7.15.1.3.1.1	pandera.checks.Check.eq	117
7.15.1.3.1.2	pandera.checks.Check.equal_to	117
7.15.1.3.1.3	pandera.checks.Check.ge	118
7.15.1.3.1.4	pandera.checks.Check.greater_than	118
7.15.1.3.1.5	pandera.checks.Check.greater_than_or_equal_to	118
7.15.1.3.1.6	pandera.checks.Check.gt	119
7.15.1.3.1.7	pandera.checks.Check.in_range	119
7.15.1.3.1.8	pandera.checks.Check.isin	119
7.15.1.3.1.9	pandera.checks.Check.le	120
7.15.1.3.1.10	pandera.checks.Check.less_than	120
7.15.1.3.1.11	pandera.checks.Check.less_than_or_equal_to	120
7.15.1.3.1.12	pandera.checks.Check.lt	121
7.15.1.3.1.13	pandera.checks.Check.ne	121
7.15.1.3.1.14	pandera.checks.Check.not_equal_to	121
7.15.1.3.1.15	pandera.checks.Check.notin	122
7.15.1.3.1.16	pandera.checks.Check.str_contains	122
7.15.1.3.1.17	pandera.checks.Check.str_endswith	122
7.15.1.3.1.18	pandera.checks.Check.str_length	123
7.15.1.3.1.19	pandera.checks.Check.str_matches	123
7.15.1.3.1.20	pandera.checks.Check.str_startswith	123
7.15.1.3.1.21	pandera.checks.Check.__call__	123
7.15.1.3.2	pandera.hypotheses.Hypothesis	124
7.15.1.3.2.1	pandera.hypotheses.Hypothesis.__init__	126
7.15.1.3.2.2	pandera.hypotheses.Hypothesis.one_sample_ttest	128
7.15.1.3.2.3	pandera.hypotheses.Hypothesis.two_sample_ttest	129
7.15.1.3.2.4	pandera.hypotheses.Hypothesis.__call__	130
7.15.2	Data Types	131
7.15.2.1	Library-agnostic dtypes	131
7.15.2.1.1	pandera.dtypes.DataType	132
7.15.2.1.1.1	pandera.dtypes.DataType.__init__	132
7.15.2.1.1.2	pandera.dtypes.DataType.check	132
7.15.2.1.1.3	pandera.dtypes.DataType.coerce	133
7.15.2.1.1.4	pandera.dtypes.DataType.try_coerce	133
7.15.2.1.1.5	pandera.dtypes.DataType.__call__	133
7.15.2.1.2	pandera.dtypes.Bool	133
7.15.2.1.2.1	pandera.dtypes.Bool.__init__	134
7.15.2.1.2.2	pandera.dtypes.Bool.check	134
7.15.2.1.2.3	pandera.dtypes.Bool.coerce	134
7.15.2.1.2.4	pandera.dtypes.Bool.try_coerce	134
7.15.2.1.2.5	pandera.dtypes.Bool.__call__	134
7.15.2.1.3	pandera.dtypes.Timestamp	134
7.15.2.1.3.1	pandera.dtypes.Timestamp.__init__	135
7.15.2.1.3.2	pandera.dtypes.Timestamp.check	135

7.15.2.1.3.3	pandera.dtypes.Timestamp.coerce	135
7.15.2.1.3.4	pandera.dtypes.Timestamp.try_coerce	135
7.15.2.1.3.5	pandera.dtypes.Timestamp.__call__	135
7.15.2.1.4	pandera.dtypes.DateTime	135
7.15.2.1.5	pandera.dtypes.Timedelta	135
7.15.2.1.5.1	pandera.dtypes.Timedelta.__init__	136
7.15.2.1.5.2	pandera.dtypes.Timedelta.check	136
7.15.2.1.5.3	pandera.dtypes.Timedelta.coerce	136
7.15.2.1.5.4	pandera.dtypes.Timedelta.try_coerce	136
7.15.2.1.5.5	pandera.dtypes.Timedelta.__call__	136
7.15.2.1.6	pandera.dtypes.Category	137
7.15.2.1.6.1	pandera.dtypes.Category.__init__	137
7.15.2.1.6.2	pandera.dtypes.Category.check	137
7.15.2.1.6.3	pandera.dtypes.Category.coerce	137
7.15.2.1.6.4	pandera.dtypes.Category.try_coerce	138
7.15.2.1.6.5	pandera.dtypes.Category.__call__	138
7.15.2.1.7	pandera.dtypes.Float	138
7.15.2.1.7.1	pandera.dtypes.Float.__init__	138
7.15.2.1.7.2	pandera.dtypes.Float.check	139
7.15.2.1.7.3	pandera.dtypes.Float.coerce	139
7.15.2.1.7.4	pandera.dtypes.Float.try_coerce	139
7.15.2.1.7.5	pandera.dtypes.Float.__call__	139
7.15.2.1.8	pandera.dtypes.Float16	139
7.15.2.1.8.1	pandera.dtypes.Float16.__init__	140
7.15.2.1.8.2	pandera.dtypes.Float16.check	140
7.15.2.1.8.3	pandera.dtypes.Float16.coerce	140
7.15.2.1.8.4	pandera.dtypes.Float16.try_coerce	140
7.15.2.1.8.5	pandera.dtypes.Float16.__call__	140
7.15.2.1.9	pandera.dtypes.Float32	140
7.15.2.1.9.1	pandera.dtypes.Float32.__init__	141
7.15.2.1.9.2	pandera.dtypes.Float32.check	141
7.15.2.1.9.3	pandera.dtypes.Float32.coerce	141
7.15.2.1.9.4	pandera.dtypes.Float32.try_coerce	141
7.15.2.1.9.5	pandera.dtypes.Float32.__call__	141
7.15.2.1.10	pandera.dtypes.Float64	141
7.15.2.1.10.1	pandera.dtypes.Float64.__init__	142
7.15.2.1.10.2	pandera.dtypes.Float64.check	142
7.15.2.1.10.3	pandera.dtypes.Float64.coerce	142
7.15.2.1.10.4	pandera.dtypes.Float64.try_coerce	142
7.15.2.1.10.5	pandera.dtypes.Float64.__call__	143
7.15.2.1.11	pandera.dtypes.Float128	143
7.15.2.1.11.1	pandera.dtypes.Float128.__init__	143
7.15.2.1.11.2	pandera.dtypes.Float128.check	143
7.15.2.1.11.3	pandera.dtypes.Float128.coerce	144
7.15.2.1.11.4	pandera.dtypes.Float128.try_coerce	144
7.15.2.1.11.5	pandera.dtypes.Float128.__call__	144
7.15.2.1.12	pandera.dtypes.Int	144
7.15.2.1.12.1	pandera.dtypes.Int.__init__	145
7.15.2.1.12.2	pandera.dtypes.Int.check	145
7.15.2.1.12.3	pandera.dtypes.Int.coerce	145
7.15.2.1.12.4	pandera.dtypes.Int.try_coerce	145
7.15.2.1.12.5	pandera.dtypes.Int.__call__	145
7.15.2.1.13	pandera.dtypes.Int8	145
7.15.2.1.13.1	pandera.dtypes.Int8.__init__	146

7.15.2.1.13.2	pandera.dtypes.Int8.check	146
7.15.2.1.13.3	pandera.dtypes.Int8.coerce	146
7.15.2.1.13.4	pandera.dtypes.Int8.try_coerce	146
7.15.2.1.13.5	pandera.dtypes.Int8.__call__	146
7.15.2.1.14	pandera.dtypes.Int16	146
7.15.2.1.14.1	pandera.dtypes.Int16.__init__	147
7.15.2.1.14.2	pandera.dtypes.Int16.check	147
7.15.2.1.14.3	pandera.dtypes.Int16.coerce	147
7.15.2.1.14.4	pandera.dtypes.Int16.try_coerce	147
7.15.2.1.14.5	pandera.dtypes.Int16.__call__	148
7.15.2.1.15	pandera.dtypes.Int32	148
7.15.2.1.15.1	pandera.dtypes.Int32.__init__	148
7.15.2.1.15.2	pandera.dtypes.Int32.check	148
7.15.2.1.15.3	pandera.dtypes.Int32.coerce	149
7.15.2.1.15.4	pandera.dtypes.Int32.try_coerce	149
7.15.2.1.15.5	pandera.dtypes.Int32.__call__	149
7.15.2.1.16	pandera.dtypes.Int64	149
7.15.2.1.16.1	pandera.dtypes.Int64.__init__	150
7.15.2.1.16.2	pandera.dtypes.Int64.check	150
7.15.2.1.16.3	pandera.dtypes.Int64.coerce	150
7.15.2.1.16.4	pandera.dtypes.Int64.try_coerce	150
7.15.2.1.16.5	pandera.dtypes.Int64.__call__	150
7.15.2.1.17	pandera.dtypes.UInt	150
7.15.2.1.17.1	pandera.dtypes.UInt.__init__	151
7.15.2.1.17.2	pandera.dtypes.UInt.check	151
7.15.2.1.17.3	pandera.dtypes.UInt.coerce	151
7.15.2.1.17.4	pandera.dtypes.UInt.try_coerce	151
7.15.2.1.17.5	pandera.dtypes.UInt.__call__	151
7.15.2.1.18	pandera.dtypes.UInt8	151
7.15.2.1.18.1	pandera.dtypes.UInt8.__init__	152
7.15.2.1.18.2	pandera.dtypes.UInt8.check	152
7.15.2.1.18.3	pandera.dtypes.UInt8.coerce	152
7.15.2.1.18.4	pandera.dtypes.UInt8.try_coerce	152
7.15.2.1.18.5	pandera.dtypes.UInt8.__call__	153
7.15.2.1.19	pandera.dtypes.UInt16	153
7.15.2.1.19.1	pandera.dtypes.UInt16.__init__	153
7.15.2.1.19.2	pandera.dtypes.UInt16.check	153
7.15.2.1.19.3	pandera.dtypes.UInt16.coerce	154
7.15.2.1.19.4	pandera.dtypes.UInt16.try_coerce	154
7.15.2.1.19.5	pandera.dtypes.UInt16.__call__	154
7.15.2.1.20	pandera.dtypes.UInt32	154
7.15.2.1.20.1	pandera.dtypes.UInt32.__init__	155
7.15.2.1.20.2	pandera.dtypes.UInt32.check	155
7.15.2.1.20.3	pandera.dtypes.UInt32.coerce	155
7.15.2.1.20.4	pandera.dtypes.UInt32.try_coerce	155
7.15.2.1.20.5	pandera.dtypes.UInt32.__call__	155
7.15.2.1.21	pandera.dtypes.UInt64	155
7.15.2.1.21.1	pandera.dtypes.UInt64.__init__	156
7.15.2.1.21.2	pandera.dtypes.UInt64.check	156
7.15.2.1.21.3	pandera.dtypes.UInt64.coerce	156
7.15.2.1.21.4	pandera.dtypes.UInt64.try_coerce	156
7.15.2.1.21.5	pandera.dtypes.UInt64.__call__	156
7.15.2.1.22	pandera.dtypes.Complex	156
7.15.2.1.22.1	pandera.dtypes.Complex.__init__	157

7.15.2.1.22.2	pandera.dtypes.Complex.check	157
7.15.2.1.22.3	pandera.dtypes.Complex.coerce	157
7.15.2.1.22.4	pandera.dtypes.Complex.try_coerce	157
7.15.2.1.22.5	pandera.dtypes.Complex.__call__	158
7.15.2.1.23	pandera.dtypes.Complex64	158
7.15.2.1.23.1	pandera.dtypes.Complex64.__init__	158
7.15.2.1.23.2	pandera.dtypes.Complex64.check	158
7.15.2.1.23.3	pandera.dtypes.Complex64.coerce	159
7.15.2.1.23.4	pandera.dtypes.Complex64.try_coerce	159
7.15.2.1.23.5	pandera.dtypes.Complex64.__call__	159
7.15.2.1.24	pandera.dtypes.Complex128	159
7.15.2.1.24.1	pandera.dtypes.Complex128.__init__	160
7.15.2.1.24.2	pandera.dtypes.Complex128.check	160
7.15.2.1.24.3	pandera.dtypes.Complex128.coerce	160
7.15.2.1.24.4	pandera.dtypes.Complex128.try_coerce	160
7.15.2.1.24.5	pandera.dtypes.Complex128.__call__	160
7.15.2.1.25	pandera.dtypes.Complex256	160
7.15.2.1.25.1	pandera.dtypes.Complex256.__init__	161
7.15.2.1.25.2	pandera.dtypes.Complex256.check	161
7.15.2.1.25.3	pandera.dtypes.Complex256.coerce	161
7.15.2.1.25.4	pandera.dtypes.Complex256.try_coerce	161
7.15.2.1.25.5	pandera.dtypes.Complex256.__call__	161
7.15.2.1.26	pandera.dtypes.String	161
7.15.2.1.26.1	pandera.dtypes.String.__init__	162
7.15.2.1.26.2	pandera.dtypes.String.check	162
7.15.2.1.26.3	pandera.dtypes.String.coerce	162
7.15.2.1.26.4	pandera.dtypes.String.try_coerce	162
7.15.2.1.26.5	pandera.dtypes.String.__call__	162
7.15.2.2	Pandas-specific Dtypes	163
7.15.2.2.1	pandera.engines.pandas_engine.BOOL	163
7.15.2.2.1.1	pandera.engines.pandas_engine.BOOL.__init__	164
7.15.2.2.1.2	pandera.engines.pandas_engine.BOOL.check	164
7.15.2.2.1.3	pandera.engines.pandas_engine.BOOL.coerce	164
7.15.2.2.1.4	pandera.engines.pandas_engine.BOOL.try_coerce	164
7.15.2.2.1.5	pandera.engines.pandas_engine.BOOL.__call__	164
7.15.2.2.2	pandera.engines.pandas_engine.INT8	164
7.15.2.2.2.1	pandera.engines.pandas_engine.INT8.__init__	165
7.15.2.2.2.2	pandera.engines.pandas_engine.INT8.check	165
7.15.2.2.2.3	pandera.engines.pandas_engine.INT8.coerce	165
7.15.2.2.2.4	pandera.engines.pandas_engine.INT8.try_coerce	165
7.15.2.2.2.5	pandera.engines.pandas_engine.INT8.__call__	165
7.15.2.2.3	pandera.engines.pandas_engine.INT16	166
7.15.2.2.3.1	pandera.engines.pandas_engine.INT16.__init__	166
7.15.2.2.3.2	pandera.engines.pandas_engine.INT16.check	166
7.15.2.2.3.3	pandera.engines.pandas_engine.INT16.coerce	166
7.15.2.2.3.4	pandera.engines.pandas_engine.INT16.try_coerce	167
7.15.2.2.3.5	pandera.engines.pandas_engine.INT16.__call__	167
7.15.2.2.4	pandera.engines.pandas_engine.INT32	167
7.15.2.2.4.1	pandera.engines.pandas_engine.INT32.__init__	168
7.15.2.2.4.2	pandera.engines.pandas_engine.INT32.check	168
7.15.2.2.4.3	pandera.engines.pandas_engine.INT32.coerce	168
7.15.2.2.4.4	pandera.engines.pandas_engine.INT32.try_coerce	168
7.15.2.2.4.5	pandera.engines.pandas_engine.INT32.__call__	168
7.15.2.2.5	pandera.engines.pandas_engine.INT64	168

7.15.2.2.5.1	pandera.engines.pandas_engine.INT64.__init__	169
7.15.2.2.5.2	pandera.engines.pandas_engine.INT64.check	169
7.15.2.2.5.3	pandera.engines.pandas_engine.INT64.coerce	169
7.15.2.2.5.4	pandera.engines.pandas_engine.INT64.try_coerce	169
7.15.2.2.5.5	pandera.engines.pandas_engine.INT64.__call__	169
7.15.2.2.6	pandera.engines.pandas_engine.UINT8	170
7.15.2.2.6.1	pandera.engines.pandas_engine.UINT8.__init__	170
7.15.2.2.6.2	pandera.engines.pandas_engine.UINT8.check	170
7.15.2.2.6.3	pandera.engines.pandas_engine.UINT8.coerce	170
7.15.2.2.6.4	pandera.engines.pandas_engine.UINT8.try_coerce	171
7.15.2.2.6.5	pandera.engines.pandas_engine.UINT8.__call__	171
7.15.2.2.7	pandera.engines.pandas_engine.UINT16	171
7.15.2.2.7.1	pandera.engines.pandas_engine.UINT16.__init__	172
7.15.2.2.7.2	pandera.engines.pandas_engine.UINT16.check	172
7.15.2.2.7.3	pandera.engines.pandas_engine.UINT16.coerce	172
7.15.2.2.7.4	pandera.engines.pandas_engine.UINT16.try_coerce	172
7.15.2.2.7.5	pandera.engines.pandas_engine.UINT16.__call__	172
7.15.2.2.8	pandera.engines.pandas_engine.UINT32	172
7.15.2.2.8.1	pandera.engines.pandas_engine.UINT32.__init__	173
7.15.2.2.8.2	pandera.engines.pandas_engine.UINT32.check	173
7.15.2.2.8.3	pandera.engines.pandas_engine.UINT32.coerce	173
7.15.2.2.8.4	pandera.engines.pandas_engine.UINT32.try_coerce	173
7.15.2.2.8.5	pandera.engines.pandas_engine.UINT32.__call__	173
7.15.2.2.9	pandera.engines.pandas_engine.UINT64	174
7.15.2.2.9.1	pandera.engines.pandas_engine.UINT64.__init__	174
7.15.2.2.9.2	pandera.engines.pandas_engine.UINT64.check	174
7.15.2.2.9.3	pandera.engines.pandas_engine.UINT64.coerce	174
7.15.2.2.9.4	pandera.engines.pandas_engine.UINT64.try_coerce	175
7.15.2.2.9.5	pandera.engines.pandas_engine.UINT64.__call__	175
7.15.2.2.10	pandera.engines.pandas_engine.STRING	175
7.15.2.2.10.1	pandera.engines.pandas_engine.STRING.__init__	176
7.15.2.2.10.2	pandera.engines.pandas_engine.STRING.check	176
7.15.2.2.10.3	pandera.engines.pandas_engine.STRING.coerce	176
7.15.2.2.10.4	pandera.engines.pandas_engine.STRING.from_parametrized_dtype	176
7.15.2.2.10.5	pandera.engines.pandas_engine.STRING.try_coerce	176
7.15.2.2.10.6	pandera.engines.pandas_engine.STRING.__call__	176
7.15.2.2.11	pandera.engines.numpy_engine.Object	176
7.15.2.2.11.1	pandera.engines.numpy_engine.Object.__init__	177
7.15.2.2.11.2	pandera.engines.numpy_engine.Object.check	177
7.15.2.2.11.3	pandera.engines.numpy_engine.Object.coerce	177
7.15.2.2.11.4	pandera.engines.numpy_engine.Object.try_coerce	177
7.15.2.2.11.5	pandera.engines.numpy_engine.Object.__call__	178
7.15.2.3	Utility functions	178
7.15.2.3.1	pandera.dtypes.is_subdtype	178
7.15.2.3.2	pandera.dtypes.is_float	179
7.15.2.3.3	pandera.dtypes.is_int	179
7.15.2.3.4	pandera.dtypes.is_uint	179
7.15.2.3.5	pandera.dtypes.is_complex	179
7.15.2.3.6	pandera.dtypes.is_numeric	179
7.15.2.3.7	pandera.dtypes.is_bool	179
7.15.2.3.8	pandera.dtypes.is_string	180
7.15.2.3.9	pandera.dtypes.is_datetime	180
7.15.2.3.10	pandera.dtypes.is_timedelta	180
7.15.2.3.11	pandera.dtypes.immutable	180

7.15.2.4	Engines	180
7.15.2.4.1	pandera.engines.engine.Engine	181
7.15.2.4.1.1	pandera.engines.engine.Engine.dtype	181
7.15.2.4.1.2	pandera.engines.engine.Engine.get_registered_dtypes	181
7.15.2.4.1.3	pandera.engines.engine.Engine.register_dtype	181
7.15.2.4.1.4	pandera.engines.engine.Engine.__call__	182
7.15.2.4.2	pandera.engines.numpy_engine.Engine	182
7.15.2.4.2.1	pandera.engines.numpy_engine.Engine.dtype	182
7.15.2.4.3	pandera.engines.pandas_engine.Engine	183
7.15.2.4.3.1	pandera.engines.pandas_engine.Engine.dtype	183
7.15.2.4.3.2	pandera.engines.pandas_engine.Engine.numpy_dtype	183
7.15.2.5	PandasDtype Enum	183
7.15.2.5.1	pandera.engines.pandas_engine.PandasDtype	183
7.15.3	Schema Models	185
7.15.3.1	Schema Model	185
7.15.3.1.1	pandera.model.SchemaModel	185
7.15.3.1.1.1	pandera.model.SchemaModel.example	186
7.15.3.1.1.2	pandera.model.SchemaModel.strategy	187
7.15.3.1.1.3	pandera.model.SchemaModel.to_schema	187
7.15.3.1.1.4	pandera.model.SchemaModel.to_yaml	187
7.15.3.1.1.5	pandera.model.SchemaModel.validate	187
7.15.3.2	Model Components	188
7.15.3.2.1	pandera.model_components.Field	189
7.15.3.2.2	pandera.model_components.check	190
7.15.3.2.3	pandera.model_components.dataframe_check	190
7.15.3.3	Typing	190
7.15.3.3.1	pandera.typing	190
7.15.3.4	Config	192
7.15.3.4.1	pandera.model.BaseConfig	192
7.15.4	Decorators	192
7.15.4.1	pandera.decorators.check_input	192
7.15.4.2	pandera.decorators.check_output	194
7.15.4.3	pandera.decorators.check_io	195
7.15.4.4	pandera.decorators.check_types	196
7.15.5	Schema Inference	196
7.15.5.1	pandera.schema_inference.infer_schema	196
7.15.6	IO Utilities	197
7.15.6.1	pandera.io.from_yaml	197
7.15.6.2	pandera.io.to_yaml	197
7.15.6.3	pandera.io.to_script	197
7.15.7	Data Synthesis Strategies	197
7.15.7.1	pandera.strategies	198
7.15.8	Extensions	204
7.15.8.1	pandera.extensions	204
7.15.9	Errors	205
7.15.9.1	pandera.errors.SchemaError	205
7.15.9.2	pandera.errors.SchemaErrors	206
7.15.9.3	pandera.errors.SchemaInitError	206
7.15.9.4	pandera.errors.SchemaDefinitionError	206
7.16	Contributing	206
7.16.1	Getting Started	206
7.16.2	Contributing to the Codebase	206
7.16.2.1	Environment Setup	206
7.16.2.1.1	Option 1: mini conda Setup	206

7.16.2.1.2	Option 2: <code>virtualenv</code> Setup	207
7.16.2.1.3	Run Tests	207
7.16.2.1.4	Build Documentation Locally	207
7.16.2.1.5	Adding New Dependencies	207
7.16.2.1.6	Set up <code>pre-commit</code>	207
7.16.2.2	Making Changes	207
7.16.2.3	Run the Full Test Suite Locally	208
7.16.2.3.1	Using <code>mamba</code> (optional)	208
7.16.2.4	Project Releases	208
7.16.2.5	Contributing Documentation	208
7.16.2.6	Contributing Bugfixes	208
7.16.2.7	Contributing Enhancements	209
7.16.2.8	Making Pull Requests	209
7.16.2.8.1	Bugfixes	209
7.16.2.8.2	Documentation	209
7.16.2.8.3	Enhancements	209
7.16.2.9	Questions, Ideas, General Discussion	209
7.16.2.10	Dataframe Schema Style Guides	209
8	How to Cite	211
8.1	Paper	211
8.2	Software Package	211
9	License and Credits	213
10	Indices and tables	215
	Python Module Index	217
	Index	219

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

pandera provides a flexible and expressive API for performing data validation on dataframes to make data processing pipelines more readable and robust.

Dataframes 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. Define a schema once and use it to validate *different dataframe types* including `pandas`, `dask`, `modin`, and `koalas`.
2. *Check* the types and properties of columns in a `pd.DataFrame` or values in a `pd.Series`.
3. Perform more complex statistical validation like *hypothesis testing*.
4. Seamlessly integrate with existing data analysis/processing pipelines via *function decorators*.
5. Define schema models with the *class-based API* with pydantic-style syntax and validate dataframes using the typing syntax.
6. *Synthesize data* from schema objects for property-based testing with pandas data structures.
7. *Lazily Validate* dataframes so that all validation rules are executed before raising an error.
8. *Integrate* with a rich ecosystem of python tools like `pydantic` and `mypy`.

INSTALL

Install with *pip*:

```
pip install pandera
```

Or conda:

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

1.1 Extras

Installing additional 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[mypy]       # enable static type-linting of pandas
pip install pandera[dask]       # validate dask dataframes
pip install pandera[koalas]     # validate koalas dataframes
pip install pandera[modin]      # validate modin dataframes
pip install pandera[modin-ray]  # validate modin dataframes with ray
pip install pandera[modin-dask] # validate modin dataframes with dask
pip install pandera[all]        # all packages
```


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 `DataType`:

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

(continues on next page)

(continued from previous page)

```
# pandas dtype string aliases
"int_column2": pa.Column("int64"),
"float_column2": pa.Column("float64"),
# pandas > 1.0.0 support native "string" type
"str_column2": pa.Column("str"),

# pandera DataType
"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 *DataType*

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)
```


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

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.

ISSUES

Submit issues, feature requests or bugfixes on [github](#).

NEED HELP?

There are many ways of getting help with your questions. You can ask a question on [Github Discussions](#) page or reach out to the maintainers and pandera community on [Discord](#)

7.1 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 `Columns` and an `Index`.

```
import pandera as pa

from pandera import Column, DataFrameSchema, Check, Index

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

You can refer to [Schema Models](#) to see how to define dataframe schemas using the alternative pydantic/dataclass-style syntax.

7.1.1 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.

Similarly to pandas, the data type can be specified as:

- a string alias, as long as it is recognized by pandas.
- a python type: *int*, *float*, *double*, *bool*, *str*
- a numpy data type
- a pandas extension type: it can be an instance (e.g *pd.CategoricalDtype(["a", "b"])*) or a class (e.g *pandas.CategoricalDtype*) if it can be initialized with default values.
- a pandera *DataType*: it can also be an instance or a class.

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.

7.1.1.1 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(float, Check(lambda x: x > 0))
})

non_null_schema.validate(df)
```

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

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

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

```

column1
0      5.0
1      1.0
2      NaN

```

7.1.1.2 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(str, 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(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(object, coerce=True, nullable=True)
})
schema_float = DataFrameSchema({
    "column1": Column(float, coerce=True, nullable=True)
})

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

```

```

column1    object
dtype: object

```

(continues on next page)

```
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)`.

7.1.1.3 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(int, required=False),
    "column2": Column(str)
})

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(int),
    "column2": Column(str),
})

schema.validate(df)
```

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

7.1.1.4 Ordered Columns

7.1.1.5 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(int, name="column1")
column2_schema = pa.Column(str, 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.

7.1.1.6 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({
```

(continues on next page)

(continued from previous page)

```

"num_var_.+": pa.Column(
    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(
        float,
        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

(continues on next page)

(continued from previous page)

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

7.1.1.7 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(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=DataType(int64)>}
```

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(str)}, strict='filter')

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

```
   column2
0    keep
1     me
```

7.1.1.8 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(int), "b": pa.Column(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
```

7.1.1.9 Validating the joint uniqueness of columns

In some cases you might want to ensure that a group of columns are unique:

```
import pandas as pd
import pandera as pa

schema = pa.DataFrameSchema(
    columns={col: pa.Column(int) for col in ["a", "b", "c"]},
    unique=["a", "c"],
)
df = pd.DataFrame.from_records([
    {"a": 1, "b": 2, "c": 3},
    {"a": 1, "b": 2, "c": 3},
])
schema.validate(df)
```

```
Traceback (most recent call last):
...
SchemaError: columns>('a', 'c') not unique:
column index failure_case
0      a      0          1
1      a      1          1
2      c      0          3
3      c      1          3
```

7.1.2 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(int)},
    index=Index(
        str,
        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
```

7.1.3 MultiIndex Validation

pandera also supports multi-index column and index validation.

7.1.3.1 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(int),
    ("foo", "baz"): Column(str)
})

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
```

7.1.3.2 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(int)},
    index=MultiIndex([
        Index(str,
              Check(lambda s: s.isin(["foo", "bar"])),
              name="index0"),
        Index(int, name="index1"),
    ])
)

df = pd.DataFrame(
```

(continues on next page)

(continued from previous page)

```

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

7.1.4 Get Pandas Data Types

Pandas provides a *dtype* parameter for casting a dataframe to a specific dtype schema. *DataFrameSchema* provides a *dtypes* property which returns a dictionary whose keys are column names and values are *DataTypes*.

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(int),
        "column2": pa.Column(pa.Category),
        "column3": pa.Column(bool)
    },
)

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

print(schema.validate(df))

```

	column1	column2	column3
a	1	valueA	True
b	1	valueB	True

7.1.5 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(int, pa.Check(lambda s: s >= 0))},
    strict=True)

transformed_schema = schema.add_columns({
    "col2": pa.Column(str, pa.Check(lambda s: s == "value")),
    "col3": pa.Column(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(int, pa.Check(lambda s: s >= 0)),
        "col2": pa.Column(str, pa.Check(lambda x: x <= 0)),
        "col3": pa.Column(object, pa.Check(lambda x: x == 0)),
    },
```

(continues on next page)

(continued from previous page)

```

    strict=True,
)

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

```

```

<Schema DataFrameSchema(
  columns={
    'col1': <Schema Column(name=col1, type=DataType(int64))>
  },
  checks=[],
  coerce=False,
  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(int),
        "column2": Column(float)
    },
    index=Index(int, name = "column3"),
    strict=True,
    coerce=True,
)
print(schema.set_index(["column1"], append = True))

```

```

<Schema DataFrameSchema(
  columns={
    'column2': <Schema Column(name=column2, type=DataType(float64))>
  },
  checks=[],
  coerce=True,
  dtype=None,
  index=<Schema MultiIndex(
    indexes=[
      <Schema Index(name=column3, type=DataType(int64))>
      <Schema Index(name=column1, type=DataType(int64))>
    ]
  )
  coerce=False,
  strict=False,
  name=None,
)

```

(continues on next page)

```

        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()`.

7.2 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 explicitly 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.

7.2.1 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"],
})

```

(continues on next page)

(continued from previous page)

```
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=DataType(int64)> 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
```

7.2.2 Validate on Initialization

new in 0.8.0

Pandera provides an interface for validating dataframes on initialization. This API uses the `pandera.typing.pandas.DataFrame` generic type to validated against the `SchemaModel` type variable on initialization:

```
import pandas as pd
import pandera as pa

from pandera.typing import DataFrame, Series

class Schema(pa.SchemaModel):
    state: Series[str]
```

(continues on next page)

(continued from previous page)

```

city: Series[str]
price: Series[int] = pa.Field(in_range={"min_value": 5, "max_value": 20})

df = DataFrame[Schema](
    {
        'state': ['NY', 'FL', 'GA', 'CA'],
        'city': ['New York', 'Miami', 'Atlanta', 'San Francisco'],
        'price': [8, 12, 10, 16],
    }
)
print(df)

```

	state	city	price
0	NY	New York	8
1	FL	Miami	12
2	GA	Atlanta	10
3	CA	San Francisco	16

Refer to *Supported DataFrame Libraries (New)* to see how this syntax applies to other supported dataframe types.

7.2.3 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=DataType(int64))>
    'month': <Schema Column(name=month, type=DataType(int64))>
    'day': <Schema Column(name=day, type=DataType(int64))>
  },
  checks=[],
  coerce=False,
  dtype=None,
  index=None,
  strict=False
  name=None,
  ordered=False
)>

```

You can also 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

Or you can use the *SchemaModel()* class directly to validate dataframes, which is syntactic sugar that simply delegates to the *validate()* method.

```
print(InputSchema(df))
```

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

7.2.4 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.

7.2.5 Supported dtypes

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

7.2.5.1 Dtype aliases

pandera.typing aliases will be deprecated in a future version, please use *DataType* subclasses instead.

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

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

7.2.5.2 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.
```

7.2.5.3 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

7.2.5.3.1 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'].
```

7.2.5.3.2 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'.

```

7.2.6 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)

```

7.2.7 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]:

```

(continues on next page)

(continued from previous page)

```

return (
    df.assign(passengers=[61000, 50000, 45000])
       .set_index(pd.Index([1, 2, 3]))
       .astype({"year": int})
)

print(transform(df))

```

	year	passengers
1	2000	61000
2	2001	50000
3	2002	45000

7.2.8 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.

7.2.9 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"

```

(continues on next page)

(continued from previous page)

```

        multiindex_strict = True
        multiindex_coerce = True

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

```

```

<Schema MultiIndex(
  indexes=[
    <Schema Index(name=year, type=DataType(int64))>
    <Schema Index(name=month, type=DataType(int64))>
  ]
  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*.

7.2.10 Custom Checks

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

7.2.10.1 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]:

```

(continues on next page)

```
    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=DataType(int64)> failed series.
↳ validator 1:
<Check check_means>
```


7.2.10.2 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)
```

7.2.10.3 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=DataType(int64)> failed element-  
wise validator 0:  
<Check foobar>  
failure cases:  
   index  failure_case  
0      0             1  
1      1             2  
2      2             3
```

7.2.11 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)  
    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
```

7.3 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(
    str,
    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
```

7.4 Checks

7.4.1 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(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(str, [
        pa.Check(lambda s: s.str.startswith("value")),
        pa.Check(lambda s: s.str.split("_", expand=True).shape[1] == 2)
    ]),
})
```

7.4.2 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(float, Check.less_than(100)),
    "one_to_three": Column(int, Check.isin([1, 2, 3])),
    "phone_number": Column(str, Check.str_matches(r'^[a-z0-9-]+$')),
})
```

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

7.4.3 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(
        int,
        checks=[
            # a vectorized check that returns a bool
            pa.Check(lambda s: s.mean() > 5, element_wise=False),

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

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

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

7.4.4 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.

7.4.5 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(
        float, [
            # groupby as a single column
            pa.Check(
                lambda g: g[False].mean() > 6,
                groupby="age_less_than_20"),

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

            # groupby as a callable with signature:
            # (DataFrame) -> DataFrameGroupBy
            pa.Check(
                lambda g: g[(False, "M")].median() == 6.75,
                groupby=lambda df: (
                    df.assign(age_less_than_15=lambda d: d["age"] < 15)
                    .groupby(["age_less_than_15", "sex"])),
            ]),
    "age": pa.Column(int, pa.Check(lambda s: s > 0)),
    "age_less_than_20": pa.Column(bool),
    "sex": pa.Column(str, 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)

```

7.4.6 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

```

(continues on next page)

(continued from previous page)

```

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

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

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(float),
        "height_B": pa.Column(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.

7.4.7 Raise `UserWarning` on Check Failure

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

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

One scenario where you'd want to do this would be in a data pipeline that does some preprocessing, checks for normality in certain columns, and writes the resulting dataset to a table. In this case, you want to see if your normality assumptions

are not fulfilled by certain columns, but you still want the resulting table for further analysis.

```
import warnings

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

from scipy.stats import normaltest

np.random.seed(1000)

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

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

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>
```


7.4.8 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.

7.5 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.

7.5.1 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(
        float, [
            Hypothesis.two_sample_ttest(
                sample1="M",
                sample2="F",
                groupby="sex",
                relationship="greater_than",
                alpha=0.05,
                equal_var=True),
        ]),
    "sex": Column(str)
})

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'
```

(continues on next page)

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

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

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

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

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

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

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

schema.validate(df)
```

7.5.2 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
```

(continues on next page)

(continued from previous page)

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

schema = DataFrameSchema({
    "height": Column(
        float, Hypothesis.two_sample_ttest(
            "A", "B",
            groupby="group",
            relationship="less_than",
            alpha=0.05
        )
    ),
    "group": Column(str, 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)
```

7.6 Pandera Data Types

new in 0.7.0

7.6.1 Motivations

Pandera defines its own interface for data types in order to abstract the specifics of dataframe-like data structures in the python ecosystem, such as Apache Spark, Apache Arrow and xarray.

Note: In the following section Pandera `Data Type` refers to a `pandera.dtypes.DataType` object whereas `native data type` refers to data types used by third-party libraries that Pandera supports (e.g. pandas).

Most of the time, it is transparent to end users since pandera columns and indexes accept native data types. However, it is possible to extend the pandera interface by:

- modifying the **data type check** performed during schema validation.
- modifying the behavior of the **coerce** argument for `DataFrameSchema`.
- adding your **own custom data types**.

7.6.2 DataType basics

All pandera data types inherit from `pandera.dtypes.DataType` and must be hashable.

A data type implements three key methods:

- `pandera.dtypes.DataType.check()` which validates that data types are equivalent.
- `pandera.dtypes.DataType.coerce()` which coerces a data container (e.g. `pandas.Series`) to the data type.
- The dunder method `__str__()` which should output the native alias. For example `str(pandera.Float64) == "float64"`

For pandera's validation methods to be aware of a data type, it has to be registered with the targeted engine via `pandera.engines.Engine.register_dtype()`. An engine is in charge of mapping a pandera `DataType` with a native data type counterpart belonging to a third-party library. The mapping can be queried with `pandera.engines.Engine.dtype()`.

As of pandera 0.7.0, only the pandas `Engine` is supported.

7.6.3 Example

Let's extend `pandas.BooleanDtype` coercion to handle the string literals "True" and "False".

```
import pandas as pd
import pandera as pa
from pandera import dtypes
from pandera.engines import pandas_engine

@pandas_engine.Engine.register_dtype # step 1
@dtypes.immutable # step 2
```

(continues on next page)

(continued from previous page)

```

class LiteralBool(pandas_engine.BOOL): # step 3
    def coerce(self, series: pd.Series) -> pd.Series:
        """Coerce a pandas.Series to date types."""
        if pd.api.types.is_string_dtype(series):
            series = series.replace({"True": 1, "False": 0})
        return series.astype("boolean")

data = pd.Series(["True", "False"], name="literal_bools")

# step 4
print(
    pa.SeriesSchema(LiteralBool(), coerce=True, name="literal_bools")
    .validate(data)
    .dtype
)

```

```
boolean
```

The example above performs the following steps:

1. Register the data type with the pandas engine.
2. `pandera.dtypes.immutable()` creates an immutable (and hashable) `dataclass()`.
3. Inherit `pandera.engines.pandas_engine.BOOL`, which is the pandera representation of `pandas.BooleanDtype`. This is not mandatory but it makes our life easier by having already implemented all the required methods.
4. Check that our new data type can coerce the string literals.

So far we did not override the default behavior:

```

import pandera as pa

pa.SeriesSchema("boolean", coerce=True).validate(data)

```

Traceback (most recent call last):

```

...
pandera.errors.SchemaError: Error while coercing 'literal_bools' to type boolean: Need
↪to pass bool-like values

```

To completely replace the default `BOOL`, we need to supply all the equivalent representations to `register_dtype()`. Behind the scenes, when `pa.SeriesSchema("boolean")` is called the corresponding pandera data type is looked up using `pandera.engines.engine.Engine.dtype()`.

```

print(f"before: {pandas_engine.Engine.dtype('boolean').__class__}")

@pandas_engine.Engine.register_dtype(
    equivalents=["boolean", pd.BooleanDtype, pd.BooleanDtype()],
)
@dtypes.immutable
class LiteralBool(pandas_engine.BOOL):

```

(continues on next page)

(continued from previous page)

```

def coerce(self, series: pd.Series) -> pd.Series:
    """Coerce a pandas.Series to date types."""
    if pd.api.types.is_string_dtype(series):
        series = series.replace({"True": 1, "False": 0})
    return series.astype("boolean")

print(f"after: {pandas_engine.Engine.dtype('boolean').__class__}")

for dtype in ["boolean", pd.BooleanDtype, pd.BooleanDtype()]:
    pa.SeriesSchema(dtype, coerce=True).validate(data)

```

```

before: <class 'pandera.engines.pandas_engine.BOOL'>
after: <class 'LiteralBool'>

```

Note: For convenience, we specified both `pd.BooleanDtype` and `pd.BooleanDtype()` as equivalents. That gives us more flexibility in what pandera schemas can recognize (see last for-loop above).

7.6.4 Parametrized data types

Some data types can be parametrized. One common example is `pandas.CategoricalDtype`.

The `equivalents` argument of `register_dtype()` does not handle this situation but will automatically register a `classmethod()` with signature `from_parametrized_dtype(cls, equivalent:...)` if the decorated `DataType` defines it. The `equivalent` argument must be type-annotated because it is leveraged to dispatch the input of `dtype` to the appropriate `from_parametrized_dtype` class method.

For example, here is a snippet from `pandera.engines.pandas_engine.Category`:

```

import pandas as pd
from pandera import dtypes

@classmethod
def from_parametrized_dtype(
    cls, cat: Union[dtypes.Category, pd.CategoricalDtype]
):
    """Convert a categorical to
    a Pandera :class:`pandera.dtypes.pandas_engine.Category`."""
    return cls(categories=cat.categories, ordered=cat.ordered) # type: ignore

```

Note: The dispatch mechanism relies on `functools.singledispatch()`. Unlike the built-in implementation, `typing.Union` is recognized.

7.7 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.

7.7.1 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(int,
                      Check(lambda x: 0 <= x <= 10, element_wise=True)),
    "column2": Column(float, Check(lambda x: x < -1.2)),
})

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

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

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

You can also provide the argument name as a string

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

Or an integer representing the index in the positional arguments.

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

7.7.2 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(int, Check(lambda x: x == 0))
})

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

```

7.7.3 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

7.7.4 Decorate Functions and Coroutines

All pandera decorators work on synchronous as well as asynchronous code, on both bound and unbound functions/coroutines. For example, one can use the same decorators on:

- sync/async functions
- sync/async methods
- sync/async class methods
- sync/async static methods

All decorators work on sync/async regular/class/static methods of metaclasses as well.

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

class Schema(pa.SchemaModel):
    col1: Series[int]

    class Config:
        strict = True

@pa.check_types
async def coroutine(df: DataFrame[Schema]) -> DataFrame[Schema]:
    return df

@pa.check_types
async def function(df: DataFrame[Schema]) -> DataFrame[Schema]:
    return df

class SomeClass:
    @pa.check_output(Schema.to_schema())
    async def regular_coroutine(self, df) -> DataFrame[Schema]:
        return df

    @classmethod
    @pa.check_input(Schema.to_schema(), "df")
    async def class_coroutine(cls, df):
        return Schema.validate(df)

    @staticmethod
    @pa.check_io(df=Schema.to_schema(), out=Schema.to_schema())
    def static_method(df):
        return df
```

7.8 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=DataType(int64))>
    'column2': <Schema Column(name=column2, type=DataType(object))>
    'column3': <Schema Column(name=column3, type=DataType(datetime64[ns]))>
  },
  checks=[],
  coerce=True,
  dtype=None,
  index=<Schema Index(name=None, type=DataType(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.

7.8.1 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.

7.8.1.1 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

schema = DataFrameSchema(
    columns={
        "column1": Column(
            dtype=pandera.engines.numpy_engine.Int64,
            checks=[
                Check.greater_than_or_equal_to(min_value=5.0),
                Check.less_than_or_equal_to(max_value=20.0),
            ],
            nullable=False,
            unique=False,
            coerce=False,
            required=True,
            regex=False,
        ),
        "column2": Column(
            dtype=pandera.engines.numpy_engine.Object,
            checks=None,
            nullable=False,
            unique=False,
            coerce=False,
            required=True,
            regex=False,
        ),
        "column3": Column(
            dtype=pandera.engines.pandas_engine.DateTime,
            checks=[
                Check.greater_than_or_equal_to(
                    min_value=Timestamp("2010-01-01 00:00:00")
                ),
                Check.less_than_or_equal_to(
                    max_value=Timestamp("2012-01-01 00:00:00")
                ),
            ],
            nullable=False,
```

(continues on next page)

(continued from previous page)

```

        unique=False,
        coerce=False,
        required=True,
        regex=False,
    ),
},
index=Index(
    dtype=pandera.engines.numpy_engine.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.

7.8.1.2 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.replace(f"{{pa.__version__}}", "{{PANDERA_VERSION}}"))

```

```

schema_type: dataframe
version: {PANDERA_VERSION}
columns:
  column1:
    dtype: int64
    nullable: false
    checks:
      greater_than_or_equal_to: 5.0
      less_than_or_equal_to: 20.0
    unique: false
    coerce: false
    required: true
    regex: false
  column2:
    dtype: object
    nullable: false
    checks: null

```

(continues on next page)

```
unique: false
coerce: false
required: true
regex: false
column3:
  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'
  unique: false
  coerce: false
  required: true
  regex: false
checks: null
index:
- 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
unique: null
```

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.

7.9 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 `data` type does not match.
- if `coerce=True`, the dataframe column cannot be coerced into the specified `data` type.
- 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(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(int),
        "float_column": Column(float, Check.greater_than(0)),
        "str_column": Column(str, Check.equal_to("a")),
        "date_column": Column(pa.DateTime),
    },
    strict=True
)

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

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

Traceback (most recent call last):

```
...
pandera.errors.SchemaErrors: A total of 5 schema errors were found.
```

Error Counts

```
-----
- column_not_in_schema: 1
- column_not_in_dataframe: 1
- schema_component_check: 3
```

(continues on next page)

```

Schema Error Summary
-----

```

schema_context	column	check	failure_cases	n_failure_cases
DataFrameSchema	<NA>	column_in_dataframe	[date_column]	1
		column_in_schema	[unknown_column]	1
Column	float_column	dtype('float64')	[int64]	1
	int_column	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:")
    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  dtype('int64')  None
3 Column          float_column  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

```

(continues on next page)

(continued from previous page)

```

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

```

7.10 Data Synthesis Strategies

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.

7.10.1 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
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.

7.10.2 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)

```

(continues on next page)

```
@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.

7.10.3 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")

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)
```

7.10.4 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.validate(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.
```

7.10.4.1 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.

7.10.4.2 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.

7.10.5 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.

7.11 Extensions

new in 0.6.0

7.11.1 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({
```

(continues on next page)

(continued from previous page)

```

    "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.

7.11.2 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(
    pandera_dtype: pa.DataType,
    strategy: Optional[st.SearchStrategy] = None,
    *,
    value,
):
    if strategy is None:
        return st.pandas_dtype_strategy(
            pandera_dtype, strategy=hypothesis.strategies.just(value),
        )
    return strategy.filter(lambda x: x == value)

```

As you may notice, the pandera strategy interface has two arguments followed by keyword-only arguments that match the check function keyword-only check statistics. The `pandera_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(
    pandera_dtype: pa.DataType,
    strategy: Optional[st.SearchStrategy] = None,
    *,
    min_value,
    max_value
):
    if strategy is None:
        return st.pandas_dtype_strategy(
            pandera_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)
```

7.11.3 Check Types

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

7.11.3.1 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.

7.11.3.2 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("xyxy"),
})

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

7.11.4 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", "col_b"])
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
```

(continues on next page)

(continued from previous page)

```

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

```

Reading Third-Party Schema

new in 0.7.0

Pandera now accepts schema from other data validation frameworks. This requires a pandera installation with the `io` extension; please see the [installation](#) instructions for more details.

7.12 Frictionless Data Schema

Note: Please see the [Frictionless schema](#) documentation for more information on this standard.

`pandera.io.from_frictionless_schema(schema)`

Create a `DataFrameSchema` from either a frictionless json/yaml schema file saved on disk, or from a frictionless schema already loaded into memory.

Each field from the frictionless schema will be converted to a pandera column specification using `FrictionlessFieldParser` to map field characteristics to pandera column specifications.

Parameters `schema` (`Union[str, Path, Dict, Schema]`) – the frictionless schema object (or a string/Path to the location on disk of a schema specification) to parse.

Return type `DataFrameSchema`

Returns dataframe schema with frictionless field specs converted to pandera column checks and constraints for use as normal.

Example

Here, we're defining a very basic frictionless schema in memory before parsing it and then querying the resulting *DataFrameSchema* object as per any other Pandera schema:

```
>>> from pandera.io import from_frictionless_schema
>>>
>>> FRICTIONLESS_SCHEMA = {
...     "fields": [
...         {
...             "name": "column_1",
...             "type": "integer",
...             "constraints": {"minimum": 10, "maximum": 99}
...         },
...         {
...             "name": "column_2",
...             "type": "string",
...             "constraints": {"maxLength": 10, "pattern": "\\S+"}
...         },
...     ],
...     "primaryKey": "column_1"
... }
>>> schema = from_frictionless_schema(FRICTIONLESS_SCHEMA)
>>> schema.columns["column_1"].checks
[<Check in_range: in_range(10, 99)>]
>>> schema.columns["column_1"].required
True
>>> schema.columns["column_1"].unique
True
>>> schema.columns["column_2"].checks
[<Check str_length: str_length(None, 10)>, <Check str_matches: str_matches(re.
↪compile('^\\S+$'))>]
```

under the hood, this uses the *FrictionlessFieldParser* class to parse each frictionless field (column):

class `pandera.io.FrictionlessFieldParser`(*field*, *primary_keys*)

Parses frictionless data schema field specifications so we can convert them to an equivalent Pandera *Column* schema.

For this implementation, we are using field names, constraints and types but leaving other frictionless parameters out (e.g. foreign keys, type formats, titles, descriptions).

Parameters

- **field** – a field object from a frictionless schema.
- **primary_keys** – the primary keys from a frictionless schema. These are used to ensure primary key fields are treated properly - no duplicates, no missing values etc.

property checks: `Optional[Dict]`

Convert a set of frictionless schema field constraints into checks.

This parses the standard set of frictionless constraints which can be found [here](#) and maps them into the equivalent pandera checks.

Return type `Optional[Dict]`

Returns a dictionary of pandera *Check* objects which capture the standard constraint logic of a frictionless schema field.

property coerce: bool

Determine whether values within this field should be coerced.

This currently returns True for all fields within a frictionless schema.

Return type bool

property dtype: str

Determine what type of field this is, so we can feed that into *DataType*. If no type is specified in the frictionless schema, we default to string values.

Return type str

Returns the pandas-compatible representation of this field type as a string.

property nullable: bool

Determine whether this field can contain missing values.

If a field is a primary key, this will return False.

Return type bool

property regex: bool

Determine whether this field name should be used for regex matches.

This currently returns False for all fields within a frictionless schema.

Return type bool

property required: bool

Determine whether this field must exist within the data.

This currently returns True for all fields within a frictionless schema.

Return type bool

to_pandera_column()

Export this field to a column spec dictionary.

Return type Dict

property unique: bool

Determine whether this field can contain duplicate values.

If a field is a primary key, this will return True.

Return type bool

7.13 Supported DataFrame Libraries (New)

Pandera started out as a pandas-specific dataframe validation library, and moving forward its core functionality will continue to support pandas. However, pandera's adoption has resulted in the realization that it can be a much more powerful tool by supporting other dataframe-like formats.

7.13.1 Scaling Up Data Validation

Pandera provides multiple ways of scaling up data validation to dataframes that don't fit into memory. Fortunately, pandera doesn't have to re-invent the wheel. Standing on shoulders of giants, it integrates with the existing ecosystem of libraries that allow you to perform validations on out-of-memory dataframes.

<i>Dask</i>	Apply pandera schemas to Dask dataframe partitions.
<i>Fugue</i>	Apply pandera schemas to distributed dataframe partitions with Fugue.
<i>Koalas</i>	A pandas drop-in replacement, distributed using a Spark backend.
<i>Modin</i>	A pandas drop-in replacement, distributed using a Ray or Dask backend.

7.13.1.1 Data Validation with Dask

new in 0.8.0

Dask is a distributed compute framework that offers a pandas-like dataframe API. You can use pandera to validate `DataFrame()` and `Series()` objects directly. First, install pandera with the dask extra:

```
pip install pandera[dask]
```

Then you can use pandera schemas to validate dask dataframes. In the example below we'll use the *class-based API* to define a `SchemaModel` for validation.

```
import dask.dataframe as dd
import pandas as pd
import pandera as pa

from pandera.typing.dask import DataFrame, Series

class Schema(pa.SchemaModel):
    state: Series[str]
    city: Series[str]
    price: Series[int] = pa.Field(in_range={"min_value": 5, "max_value": 20})

ddf = dd.from_pandas(
    pd.DataFrame(
        {
            'state': ['FL', 'FL', 'FL', 'CA', 'CA', 'CA'],
            'city': [
                'Orlando',
                'Miami',
                'Tampa',
                'San Francisco',
                'Los Angeles',
                'San Diego',
            ],
            'price': [8, 12, 10, 16, 20, 18],
        }
    ),
    npartitions=2
)
```

(continues on next page)

(continued from previous page)

```
pandera_ddf = Schema(ddf)
print(pandera_ddf)
```

```
Dask DataFrame Structure:
                state  city  price
npartitions=2
0              object  object  int64
3                ...    ...    ...
5                ...    ...    ...
Dask Name: validate, 4 tasks
```

As you can see, passing the dask dataframe into `Schema` will produce another dask dataframe which hasn't been evaluated yet. What this means is that pandera will only validate when the dask graph is evaluated.

```
print(pandera_ddf.compute())
```

```
state      city  price
0  FL      Orlando    8
1  FL      Miami     12
2  FL      Tampa     10
3  CA  San Francisco  16
4  CA  Los Angeles   20
5  CA  San Diego     18
```

You can also use the `check_types()` decorator to validate dask dataframes at runtime:

```
@pa.check_types
def function(ddf: DataFrame[Schema]) -> DataFrame[Schema]:
    return ddf[ddf["state"] == "CA"]
print(function(ddf).compute())
```

```
state      city  price
3  CA  San Francisco  16
4  CA  Los Angeles   20
5  CA  San Diego     18
```

And of course, you can use the object-based API to validate dask dataframes:

```
schema = pa.DataFrameSchema({
    "state": pa.Column(str),
    "city": pa.Column(str),
    "price": pa.Column(int, pa.Check.in_range(min_value=5, max_value=20))
})
print(schema(ddf).compute())
```

```
state      city  price
0  FL      Orlando    8
1  FL      Miami     12
2  FL      Tampa     10
```

(continues on next page)

(continued from previous page)

3	CA	San Francisco	16
4	CA	Los Angeles	20
5	CA	San Diego	18

7.13.1.2 Data Validation with Fugue

Validation on big data comes in two forms. The first is performing one set of validations on data that doesn't fit in memory. The second happens when a large dataset is comprised of multiple groups that require different validations. In pandas semantics, this would be the equivalent of a `groupby-validate` operation. This section will cover using `pandera` for both of these scenarios.

`Pandera` only supports pandas `DataFrames` at the moment. However, the same `pandera` code can be used on top of Spark or Dask engines with `Fugue`. These computation engines allow validation to be performed in a distributed setting. `Fugue` is an open source abstraction layer that ports Python, pandas, and SQL code to Spark and Dask.

7.13.1.2.1 What is Fugue?

`Fugue` serves as an interface to distributed computing. Because of its non-invasive design, existing Python code can be scaled to a distributed setting without significant changes.

To run the example, `Fugue` needs to be installed separately. Using `pip`:

```
pip install fugue[spark]
```

This will also install `PySpark` because of the `spark` extra. `Dask` is available with the `dask` extra.

7.13.1.2.2 Example

In this example, a pandas `DataFrame` is created with `state`, `city` and `price` columns. `Pandera` will be used to validate that the `price` column values are within a certain range.

```
import pandas as pd

data = pd.DataFrame(
    {
        'state': ['FL', 'FL', 'FL', 'CA', 'CA', 'CA'],
        'city': [
            'Orlando', 'Miami', 'Tampa', 'San Francisco', 'Los Angeles', 'San Diego'
        ],
        'price': [8, 12, 10, 16, 20, 18],
    }
)

print(data)
```

	state	city	price
0	FL	Orlando	8
1	FL	Miami	12
2	FL	Tampa	10
3	CA	San Francisco	16

(continues on next page)

(continued from previous page)

4	CA	Los Angeles	20
5	CA	San Diego	18

Validation is then applied using pandera. A `price_validation` function is created that runs the validation. None of this will be new.

```
from pandera import Column, DataFrameSchema, Check

price_check = DataFrameSchema(
    {"price": Column(int, Check.in_range(min_value=5,max_value=20))}
)

def price_validation(data:pd.DataFrame) -> pd.DataFrame:
    return price_check.validate(data)
```

The transform function in Fugue is the easiest way to use Fugue with existing Python functions as seen in the following code snippet. The first two arguments are the `DataFrame` and function to apply. The keyword argument `schema` is required because schema is strictly enforced in distributed settings. Here, the `schema` is simply `*` because no new columns are added.

The last part of the transform function is the engine. Here, the `SparkExecutionEngine` is used to run the code on top of Spark. Fugue also has a `DaskExecutionEngine`, and passing nothing uses the default pandas-based `ExecutionEngine`. Because the `SparkExecutionEngine` is used, the result becomes a Spark `DataFrame`.

```
from fugue import transform
from fugue_spark import SparkExecutionEngine

spark_df = transform(data, price_validation, schema="*", engine=SparkExecutionEngine)
spark_df.show()
```

```
+-----+-----+-----+
|state|      city|price|
+-----+-----+-----+
|  FL|    Orlando|   8|
|  FL|     Miami|  12|
|  FL|     Tampa|  10|
|  CA|San Francisco| 16|
|  CA|  Los Angeles| 20|
|  CA|   San Diego| 18|
+-----+-----+-----+
```

7.13.1.2.3 Validation by Partition

There is an interesting use case that arises with bigger datasets. Frequently, there are logical groupings of data that require different validations. In the earlier sample data, the price range for the records with state FL is lower than the range for the state CA. Two `DataFrameSchema` will be created to reflect this. Notice their ranges for the `Check` differ.

```
price_check_FL = DataFrameSchema({
    "price": Column(int, Check.in_range(min_value=7,max_value=13)),
})
```

(continues on next page)

```
price_check_CA = DataFrameSchema({
    "price": Column(int, Check.in_range(min_value=15,max_value=21)),
})

price_checks = {'CA': price_check_CA, 'FL': price_check_FL}
```

A slight modification is needed to our `price_validation` function. Fugue will partition the whole dataset into multiple pandas DataFrames. Think of this as a `groupby`. By the time `price_validation` is used, it only contains the data for one state. The appropriate `DataFrameSchema` is pulled and then applied.

To partition our data by state, all we need to do is pass it into the `transform` function through the `partition` argument. This splits up the data across different workers before they each run the `price_validation` function. Again, this is like a `groupby-validation`.

```
def price_validation(df:pd.DataFrame) -> pd.DataFrame:
    location = df['state'].iloc[0]
    check = price_checks[location]
    check.validate(df)
    return df

spark_df = transform(data,
    price_validation,
    schema="*",
    partition=dict(by="state"),
    engine=SparkExecutionEngine)

spark_df.show()
```

```
SparkDataFrame
state:str|city:str                                |price:long
-----+-----+-----
CA      |San Francisco                                |16
CA      |Los Angeles                                  |20
CA      |San Diego                                    |18
FL      |Orlando                                       |8
FL      |Miami                                         |12
FL      |Tampa                                         |10
Total count: 6
```

Note: Because operations in a distributed setting are applied per partition, statistical validators will be applied on each partition rather than the global dataset. If no partitioning scheme is specified, Spark and Dask use default partitions. Be careful about using operations like `mean`, `min`, and `max` without partitioning beforehand.

All row-wise validations scale well with this set-up.

7.13.1.3 Data Validation with Koalas

new in 0.8.0

Koalas is a distributed compute framework that offers a pandas drop-in replacement dataframe implementation. You can use pandera to validate DataFrame() and Series() objects directly. First, install pandera with the `dask extra`:

```
pip install pandera[koalas]
```

Then you can use pandera schemas to validate koalas dataframes. In the example below we'll use the *class-based API* to define a SchemaModel for validation.

```
import databricks.koalas as ks
import pandas as pd
import pandera as pa

from pandera.typing.koalas import DataFrame, Series

class Schema(pa.SchemaModel):
    state: Series[str]
    city: Series[str]
    price: Series[int] = pa.Field(in_range={"min_value": 5, "max_value": 20})

# create a koalas dataframe that's validated on object initialization
df = DataFrame[Schema](
    {
        'state': ['FL', 'FL', 'FL', 'CA', 'CA', 'CA'],
        'city': [
            'Orlando',
            'Miami',
            'Tampa',
            'San Francisco',
            'Los Angeles',
            'San Diego',
        ],
        'price': [8, 12, 10, 16, 20, 18],
    }
)
print(df)
```

	state	city	price
0	FL	Orlando	8
1	FL	Miami	12
2	FL	Tampa	10
3	CA	San Francisco	16
4	CA	Los Angeles	20
5	CA	San Diego	18

You can also use the `check_types()` decorator to validate koalas dataframes at runtime:

```
@pa.check_types
def function(df: DataFrame[Schema]) -> DataFrame[Schema]:
```

(continues on next page)

(continued from previous page)

```

return df[df["state"] == "CA"]

print(function(df))

```

```

state      city  price
3  CA  San Francisco    16
4  CA   Los Angeles    20
5  CA   San Diego     18

```

And of course, you can use the object-based API to validate dask dataframes:

```

schema = pa.DataFrameSchema({
    "state": pa.Column(str),
    "city": pa.Column(str),
    "price": pa.Column(int, pa.Check.in_range(min_value=5, max_value=20))
})
print(schema(df))

```

```

state      city  price
0  FL      Orlando     8
1  FL      Miami     12
2  FL      Tampa     10
3  CA  San Francisco    16
4  CA   Los Angeles    20
5  CA   San Diego     18

```

7.13.1.4 Data Validation with Modin

new in 0.8.0

Modin is a distributed compute framework that offers a pandas drop-in replacement dataframe implementation. You can use pandera to validate `DataFrame()` and `Series()` objects directly. First, install pandera with the dask extra:

```

pip install pandera[modin]           # installs both ray and dask backends
pip install pandera[modin-ray]      # only ray backend
pip install pandera[modin-dask]     # only dask backend

```

Then you can use pandera schemas to validate modin dataframes. In the example below we'll use the *class-based API* to define a `SchemaModel` for validation.

```

import modin.pandas as pd
import pandas as pd
import pandera as pa

from pandera.typing.modin import DataFrame, Series

class Schema(pa.SchemaModel):
    state: Series[str]
    city: Series[str]
    price: Series[int] = pa.Field(in_range={"min_value": 5, "max_value": 20})

```

(continues on next page)

(continued from previous page)

```
# create a modin dataframe that's validated on object initialization
df = DataFrame[Schema](
    {
        'state': ['FL', 'FL', 'FL', 'CA', 'CA', 'CA'],
        'city': [
            'Orlando',
            'Miami',
            'Tampa',
            'San Francisco',
            'Los Angeles',
            'San Diego',
        ],
        'price': [8, 12, 10, 16, 20, 18],
    }
)
print(df)
```

	state	city	price
0	FL	Orlando	8
1	FL	Miami	12
2	FL	Tampa	10
3	CA	San Francisco	16
4	CA	Los Angeles	20
5	CA	San Diego	18

You can also use the `check_types()` decorator to validate modin dataframes at runtime:

```
@pa.check_types
def function(df: DataFrame[Schema]) -> DataFrame[Schema]:
    return df[df["state"] == "CA"]

print(function(df))
```

	state	city	price
3	CA	San Francisco	16
4	CA	Los Angeles	20
5	CA	San Diego	18

And of course, you can use the object-based API to validate dask dataframes:

```
schema = pa.DataFrameSchema({
    "state": pa.Column(str),
    "city": pa.Column(str),
    "price": pa.Column(int, pa.Check.in_range(min_value=5, max_value=20))
})
print(schema(df))
```

	state	city	price
0	FL	Orlando	8
1	FL	Miami	12

(continues on next page)

(continued from previous page)

2	FL	Tampa	10
3	CA	San Francisco	16
4	CA	Los Angeles	20
5	CA	San Diego	18

Note: Don't see a library that you want supported? Check out the [github issues](#) to see if that library is in the roadmap. If it isn't, open up a [new issue](#) to add support for it!

7.14 Integrations

Pandera ships with integrations with other tools in the Python ecosystem, with the goal of interoperating with libraries that you know and love.

<i>Hypothesis</i>	Use the hypothesis library to generate valid data under your schema's constraints.
<i>Pydantic</i>	Use pandera SchemaModels when defining your pydantic BaseModel
<i>Mypy</i>	Type-lint your pandas and pandera code with mypy for static type safety [experimental]

7.14.1 Pydantic

new in 0.8.0

`SchemaModel` is fully compatible with pydantic.

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

class SimpleSchema(pa.SchemaModel):
    str_col: Series[str] = pa.Field(unique=True)

class PydanticModel(pydantic.BaseModel):
    x: int
    df: DataFrame[SimpleSchema]

valid_df = pd.DataFrame({"str_col": ["hello", "world"]})
PydanticModel(x=1, df=valid_df)

invalid_df = pd.DataFrame({"str_col": ["hello", "hello"]})
PydanticModel(x=1, df=invalid_df)
```

Traceback (most recent call last):

...

(continues on next page)

(continued from previous page)

```

ValidationError: 1 validation error for PydanticModel
df
series 'str_col' contains duplicate values:
1    hello
Name: str_col, dtype: object (type=value_error)

```

Other pandera components are also compatible with pydantic:

- [SchemaModel](#)
- [DataFrameSchema](#)
- [SeriesSchema](#)
- [MultiIndex](#)
- [Column](#)
- [Index](#)

7.14.2 Mypy

new in 0.8.0

Pandera integrates with mypy to provide static type-linting of dataframes, relying on [pandas-stubs](#) for typing information.

```
pip install pandera[mypy]
```

Then enable the plugin in your `mypy.ini` or `setug.cfg` file:

```
[mypy]
plugins = pandera.mypy
```

Note: Mypy static type-linting is supported for only pandas dataframes.

Warning: This functionality is experimental . Since the [pandas-stubs](#) type stub annotations don't always match the official [pandas effort to support type annotations](#)), installing the `pandera[mypy]` extra may yield false positives in your pandas code, many of which are documented in `tests/mypy/modules`.

We encourage beta users to [file an issue](#) if they find any false positives or negatives being reported by mypy. A list of such issues can be found [here](#).

In the example below, we define a few schemas to see how type-linting with pandera works.

```

from typing import cast

import pandas as pd

import pandera as pa
from pandera.typing import DataFrame, Series

```

(continues on next page)

```

class Schema(pa.SchemaModel):
    id: Series[int]
    name: Series[str]

class SchemaOut(pa.SchemaModel):
    age: Series[int]

class AnotherSchema(pa.SchemaModel):
    id: Series[int]
    first_name: Series[str]

```

The mypy linter will complain if the output type of the function body doesn't match the function's return signature.

```

def fn(df: DataFrame[Schema]) -> DataFrame[SchemaOut]:
    return df.assign(age=30).pipe(DataFrame[SchemaOut]) # mypy okay

def fn_pipe_incorrect_type(df: DataFrame[Schema]) -> DataFrame[SchemaOut]:
    return df.assign(age=30).pipe(DataFrame[AnotherSchema]) # mypy error
    # error: Argument 1 to "pipe" of "NDFrame" has incompatible type
    # "Type[DataFrame[Any]]"; # noqa
    # expected "Union[Callable[... , DataFrame[SchemaOut]], Tuple[Callable[... ,
    # DataFrame[SchemaOut]], str]]" [arg-type] # noqa

def fn_assign_copy(df: DataFrame[Schema]) -> DataFrame[SchemaOut]:
    return df.assign(age=30) # mypy error
    # error: Incompatible return value type (got "pandas.core.frame.DataFrame",
    # expected "pandera.typing.pandas.DataFrame[SchemaOut]") [return-value]

```

It'll also complain if the input type doesn't match the expected input type. Note that we're using the `pandera.typing.pandas.DataFrame` generic type to define dataframes that are validated against the `SchemaModel` type variable on initialization.

```

schema_df = DataFrame[Schema]({"id": [1], "name": ["foo"]})
pandas_df = pd.DataFrame({"id": [1], "name": ["foo"]})
another_df = DataFrame[AnotherSchema]({"id": [1], "first_name": ["foo"]})

fn(schema_df) # mypy okay

fn(pandas_df) # mypy error
# error: Argument 1 to "fn" has incompatible type "pandas.core.frame.DataFrame"; # noqa
# expected "pandera.typing.pandas.DataFrame[Schema]" [arg-type]

fn(another_df) # mypy error
# error: Argument 1 to "fn" has incompatible type "DataFrame[AnotherSchema]";
# expected "DataFrame[Schema]" [arg-type]

```

To make mypy happy with respect to the return type, you can either initialize a dataframe of the expected type:

```
def fn_pipe_dataframe(df: DataFrame[Schema]) -> DataFrame[SchemaOut]:
    return df.assign(age=30).pipe(DataFrame[SchemaOut]) # mypy okay
```

Note: If you use the approach above with the `check_types()` decorator, pandera will do its best to not to validate the dataframe twice if it's already been initialized with the `DataFrame[Schema]**data` syntax.

Or use `typing.cast()` to indicate to mypy that the return value of the function is of the correct type.

```
def fn_cast_dataframe(df: DataFrame[Schema]) -> DataFrame[SchemaOut]:
    return cast(DataFrame[SchemaOut], df.assign(age=30)) # mypy okay
```

7.14.2.1 Limitations

An important caveat to static type-linting with pandera dataframe types is that, since pandas dataframes are mutable objects, there's no way for mypy to know whether a mutated instance of a `SchemaModel`-typed dataframe has the correct contents. Fortunately, we can simply rely on the `check_types()` decorator to verify that the output dataframe is valid.

Consider the examples below:

```
def fn_pipe_dataframe(df: DataFrame[Schema]) -> DataFrame[SchemaOut]:
    return df.assign(age=30).pipe(DataFrame[SchemaOut]) # mypy okay

def fn_cast_dataframe(df: DataFrame[Schema]) -> DataFrame[SchemaOut]:
    return cast(DataFrame[SchemaOut], df.assign(age=30)) # mypy okay

@pa.check_types
def fn_mutate_inplace(df: DataFrame[Schema]) -> DataFrame[SchemaOut]:
    out = df.assign(age=30).pipe(DataFrame[SchemaOut])
    out.drop(["age"], axis=1, inplace=True)
    return out # okay for mypy, pandera raises error

@pa.check_types
def fn_assign_and_get_index(df: DataFrame[Schema]) -> DataFrame[SchemaOut]:
    return df.assign(foo=30).iloc[:3] # okay for mypy, pandera raises error
```

Even though the outputs of these functions are incorrect, mypy doesn't catch the error during static type-linting but pandera will raise a `SchemaError` or `SchemaErrors` exception at runtime, depending on whether you're doing *lazy validation* or not.

```
@pa.check_types
def fn_cast_dataframe_invalid(df: DataFrame[Schema]) -> DataFrame[SchemaOut]:
    return cast(
        DataFrame[SchemaOut], df
    ) # okay for mypy, pandera raises error
```

Note: Don't see a library that you want supported? Check out the [github issues](#) to see if that library is in the roadmap. If it isn't, open up a [new issue](#) to add support for it!

7.15 API

<i>Core</i>	The core objects for defining pandera schemas
<i>Data Types</i>	Data types for type checking and coercion.
<i>Schema Models</i>	Alternative class-based API for defining pandera schemas.
<i>Decorators</i>	Decorators for integrating pandera schemas with python functions.
<i>Schema Inference</i>	Bootstrap schemas from real data
<i>IO Utilities</i>	Utility functions for reading/writing schemas
<i>Data Synthesis Strategies</i>	Module of functions for generating data from schemas.
<i>Extensions</i>	Utility functions for extending pandera functionality
<i>Errors</i>	Pandera-specific exceptions

7.15.1 Core

7.15.1.1 Schemas

<code>pandera.schemas.DataFrameSchema</code>	A light-weight pandas DataFrame validator.
<code>pandera.schemas.SeriesSchema</code>	Series validator.

7.15.1.1.1 `pandera.schemas.DataFrameSchema`

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

A light-weight pandas DataFrame validator.

Initialize DataFrameSchema validator.

Parameters

- **columns** (*mapping of column names and column schema component.*) – a dict where keys are column names and values are Column objects specifying the datatypes and properties of a particular column.
- **checks** (*CheckList*) – dataframe-wide checks.
- **index** – specify the datatypes and properties of the index.
- **dtype** (*PandasDtypeInputTypes*) – 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** (*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.

Warning: This feature is deprecated and no longer has an effect on validated dataframes.

- **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.
- **pandas_dtype** (*PandasDtypeInputTypes*) – alias of `dtype` for backwards compatibility.

Warning: This option will be deprecated in 0.8.0

- **unique** (*Optional[Union[str, List[str]]]*) – a list of columns that should be jointly unique.

Raises

- **SchemaInitError** – if impossible to build schema from parameters
- **SchemaInitError** – if `dtype` and `pandas_dtype` are both supplied.

Examples

```
>>> import pandera as pa
>>>
>>> schema = pa.DataFrameSchema({
...     "str_column": pa.Column(str),
...     "float_column": pa.Column(float),
...     "int_column": pa.Column(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(
...         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(
...         str, [
...             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

<code>coerce</code>	Whether to coerce series to specified type.
<code>dtype</code>	Get the dtype property.
<code>dtypes</code>	A dict where the keys are column names and values are <i>DataTypes</i> for the column.
<code>ordered</code>	Whether or not to validate the columns order.
<code>unique</code>	List of columns that should be jointly unique.

Methods

<code>__init__</code>	Initialize DataFrameSchema validator.
<code>add_columns</code>	Create a copy of the <i>DataFrameSchema</i> with extra columns.
<code>coerce_dtype</code>	Coerce dataframe to the type specified in dtype.
<code>example</code>	Generate an example of a particular size.
<code>from_yaml</code>	Create DataFrameSchema from yaml file.
<code>get_dtypes</code>	Same as the dtype property, but expands columns where <code>regex == True</code> based on the supplied dataframe.
<code>remove_columns</code>	Removes columns from a <i>DataFrameSchema</i> and returns a new copy.
<code>rename_columns</code>	Rename columns using a dictionary of key-value pairs.
<code>reset_index</code>	A method for resetting the Index of a <i>DataFrameSchema</i>
<code>select_columns</code>	Select subset of columns in the schema.
<code>set_index</code>	A method for setting the Index of a <i>DataFrameSchema</i> , via an existing Column or list of columns.
<code>strategy</code>	Create a hypothesis strategy for generating a DataFrame.
<code>to_script</code>	Create DataFrameSchema from yaml file.
<code>to_yaml</code>	Write DataFrameSchema to yaml file.
<code>update_column</code>	Create copy of a <i>DataFrameSchema</i> with updated column properties.
<code>update_columns</code>	Create copy of a <i>DataFrameSchema</i> with updated column properties.
<code>validate</code>	Check if all columns in a dataframe have a column in the Schema.
<code>__call__</code>	Alias for <i>DataFrameSchema.validate()</i> method.

7.15.1.1.1.1 pandera.schemas.DataFrameSchema.__init__

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

Initialize DataFrameSchema validator.

Parameters

- **columns** (*mapping of column names and column schema component.*) – a dict where keys are column names and values are Column objects specifying the datatypes and properties of a particular column.
- **checks** (*CheckList*) – dataframe-wide checks.
- **index** – specify the datatypes and properties of the index.
- **dtype** (*PandasDtypeInputTypes*) – 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** (*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.

Warning: This feature is deprecated and no longer has an effect on validated dataframes.

- **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.
- **pandas_dtype** (*PandasDtypeInputTypes*) – alias of `dtype` for backwards compatibility.

Warning: This option will be deprecated in 0.8.0

- **unique** (*Optional[Union[str, List[str]]]*) – a list of columns that should be jointly unique.

Raises

- **SchemaInitError** – if impossible to build schema from parameters
- **SchemaInitError** – if `dtype` and `pandas_dtype` are both supplied.

Examples

```
>>> import pandera as pa
>>>
>>> schema = pa.DataFrameSchema({
...     "str_column": pa.Column(str),
...     "float_column": pa.Column(float),
...     "int_column": pa.Column(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(
...         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(
...         str, [
...             pa.Check(lambda s: s.isin(["dog", "cat", "duck"])),
...             pa.Check(lambda s: (s == "dog").mean() > 0.5),
...         ]),
... })
```

See [here](#) for more usage details.

7.15.1.1.1.2 `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 `ForwardRef`

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(str),
...         "probability": pa.Column(float),
...     }
... )
>>> print(
...     example_schema.add_columns({"even_number": pa.Column(pa.Bool)})
... )
<Schema DataFrameSchema(
```

(continues on next page)

(continued from previous page)

```

columns={
    'category': <Schema Column(name=category, type=DataType(str))>
    'probability': <Schema Column(name=probability, type=DataType(float64))>
    'even_number': <Schema Column(name=even_number, type=DataType(bool))>
},
checks=[],
coerce=False,
dtype=None,
index=None,
strict=False
name=None,
ordered=False
)>

```

See also:`remove_columns()`**7.15.1.1.1.3 pandera.schemas.DataFrameSchema.coerce_dtype**`DataFrameSchema.coerce_dtype(obj)`

Coerce dataframe to the type specified in dtype.

Parameters `obj` (`DataFrame`) – dataframe to coerce.**Return type** `DataFrame`**Returns** dataframe with coerced dtypes**7.15.1.1.1.4 pandera.schemas.DataFrameSchema.example**`DataFrameSchema.example(size=None, n_regex_columns=1)`

Generate an example of a particular size.

Parameters `size` (`Optional[int]`) – number of elements in the generated DataFrame.**Return type** `DataFrame`**Returns** pandas DataFrame object.**7.15.1.1.1.5 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** `ForwardRef`**Returns** dataframe schema.

7.15.1.1.1.6 `pandera.schemas.DataFrameSchema.get_dtypes`

`DataFrameSchema.get_dtypes(dataframe)`

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

Return type `Dict[str, DataType]`

Returns dictionary of columns and their associated dtypes.

7.15.1.1.1.7 `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 `ForwardRef`

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(str),
...         "probability": pa.Column(float)
...     }
... )
>>>
>>> print(example_schema.remove_columns(["category"]))
<Schema DataFrameSchema(
  columns={
    'probability': <Schema Column(name=probability, type=DataType(float64))>
  },
  checks=[],
  coerce=False,
  dtype=None,
  index=None,
  strict=False
  name=None,
  ordered=False
)>
```

See also:

[`add_columns\(\)`](#)

7.15.1.1.1.8 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 `ForwardRef`

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(str),
...     "probability": pa.Column(float)
... })
>>>
>>> print(
...     example_schema.rename_columns({
...         "category": "categories",
...         "probability": "probabilities"
...     })
... )
<Schema DataFrameSchema(
  columns={
    'categories': <Schema Column(name=categories, type=DataType(str))>
    'probabilities': <Schema Column(name=probabilities,
↪ type=DataType(float64))>
  },
  checks=[],
  coerce=False,
  dtype=None,
  index=None,
  strict=False
  name=None,
  ordered=False
)>
```

See also:

[update_column\(\)](#)

7.15.1.1.1.9 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 `ForwardRef`

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(float)},
...     index = pa.Index(name="unique_id", dtype=int)
... )
>>>
>>> print(example_schema.reset_index())
<Schema DataFrameSchema(
  columns={
    'probability': <Schema Column(name=probability, type=DataType(float64))>
    'unique_id': <Schema Column(name=unique_id, type=DataType(int64))>
  },
  checks=[],
  coerce=False,
  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(str)},
...     index = pa.MultiIndex([
...         pa.Index(name="unique_id1", dtype=int),
...         pa.Index(name="unique_id2", dtype=str)
...     ])
... )
```

(continues on next page)

(continued from previous page)

```

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

```

See also:`set_index()`**7.15.1.1.10 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[Any]`) – list of column names to select.**Return type** `ForwardRef`**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(str),
...     "probability": pa.Column(float)
... })
>>>
>>> print(example_schema.select_columns(['category']))
<Schema DataFrameSchema(
  columns={
    'category': <Schema Column(name=category, type=DataType(str))>
  },
  checks=[],
  coerce=False,
  dtype=None,
  index=None,

```

(continues on next page)

```

strict=False
name=None,
ordered=False
)>

```

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

7.15.1.1.1.11 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 `ForwardRef`

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(str),
...     "probability": pa.Column(float)})
>>>
>>> print(example_schema.set_index(['category']))
<Schema DataFrameSchema(
  columns={
    'probability': <Schema Column(name=probability, type=DataType(float64))>
  },
  checks=[],
  coerce=False,
  dtype=None,
  index=<Schema Index(name=category, type=DataType(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(str),
...         "column2": pa.Column(int)
...     },
...     index=pa.Index(name = "column3", dtype = int)
... )
>>>
>>> print(example_schema.set_index(["column2"], append = True))
<Schema DataFrameSchema(
  columns={
    'column1': <Schema Column(name=column1, type=DataType(str))>
  },
  checks=[],
  coerce=False,
  dtype=None,
  index=<Schema MultiIndex(
    indexes=[
      <Schema Index(name=column3, type=DataType(int64))>
      <Schema Index(name=column2, type=DataType(int64))>
    ]
    coerce=False,
    strict=False,
    name=None,
    ordered=True
  )>,
  strict=False,
  name=None,
  ordered=False
)>

```

See also:

`reset_index()`

7.15.1.1.12 `pandera.schemas.DataFrameSchema.strategy`

`DataFrameSchema.strategy(*, size=None, n_regex_columns=1)`

Create a hypothesis strategy for generating a DataFrame.

Parameters

- **size** (`Optional[int]`) – number of elements to generate
- **n_regex_columns** (`int`) – number of regex columns to generate.

Returns a strategy that generates pandas DataFrame objects.

7.15.1.1.1.13 `pandera.schemas.DataFrameSchema.to_script`

`DataFrameSchema.to_script(fp=None)`

Create `DataFrameSchema` from yaml file.

Parameters `path` – str, Path to write script

Return type `ForwardRef`

Returns dataframe schema.

7.15.1.1.1.14 `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.

7.15.1.1.1.15 `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 `ForwardRef`

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(str),
...     "probability": pa.Column(float)
... })
>>> print(
...     example_schema.update_column(
...         'category', dtype=pa.Category
...     )
... )
<Schema DataFrameSchema(
  columns={
    'category': <Schema Column(name=category, type=DataType(category))>
    'probability': <Schema Column(name=probability, type=DataType(float64))>
  },
```

(continues on next page)

(continued from previous page)

```

checks=[],
coerce=False,
dtype=None,
index=None,
strict=False
name=None,
ordered=False
)>

```

See also:

`rename_columns()`

7.15.1.1.16 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 `ForwardRef`

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(str),
...     "probability": pa.Column(float)
... })
>>>
>>> print(
...     example_schema.update_columns(
...         {"category": {"dtype": pa.Category}}
...     )
... )
<Schema DataFrameSchema(
  columns={
    'category': <Schema Column(name=category, type=DataType(category))>
    'probability': <Schema Column(name=probability, type=DataType(float64))>
  },
  checks=[],
  coerce=False,
  dtype=None,
  index=None,
  strict=False
  name=None,
  ordered=False
)>

```

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

7.15.1.1.1.17 `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

- **check_obj** (*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(
...         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(
...         str, [
...             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

```

7.15.1.1.18 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.

7.15.1.1.2 pandera.schemas.SeriesSchema

`class pandera.schemas.SeriesSchema(dtype=None, checks=None, index=None, nullable=False, unique=False, allow_duplicates=None, coerce=False, name=None, pandas_dtype=None)`

Series validator.

Initialize series schema base object.

Parameters

- **dtype** (`Union[str, type, DataType, 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** (`Union[Check, Hypothesis, List[Union[Check, Hypothesis]], None]`) – 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.
- **unique** (bool) – Whether or not column can contain duplicate values.
- **allow_duplicates** (Optional[bool]) – Whether or not column can contain duplicate values.

Warning: This option will be deprecated in 0.8.0. Use the unique argument instead.

Parameters

- **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.
- **name** (Optional[str]) – series name.
- **pandas_dtype** (Union[str, type, DataType, ExtensionDtype, dtype, None]) – alias of dtype for backwards compatibility.

Warning: This option will be deprecated in 0.8.0

Attributes

allow_duplicates	Whether to allow duplicate values.
checks	Return list of checks or hypotheses.
coerce	Whether to coerce series to specified type.
dtype	Get the pandas dtype
name	Get SeriesSchema name.
nullable	Whether the series is nullable.
unique	Whether to check for duplicates in check object

Methods

<code>__init__</code>	Initialize series schema base object.
<code>validate</code>	Validate a Series object.
<code>__call__</code>	Alias for <code>SeriesSchema.validate()</code> method.

7.15.1.1.2.1 pandera.schemas.SeriesSchema.__init__

`SeriesSchema.__init__(dtype=None, checks=None, index=None, nullable=False, unique=False, allow_duplicates=None, coerce=False, name=None, pandas_dtype=None)`

Initialize series schema base object.

Parameters

- **dtype** (`Union[str, type, DataType, 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** (`Union[Check, Hypothesis, List[Union[Check, Hypothesis]], None]`) – 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.
- **unique** (`bool`) – Whether or not column can contain duplicate values.
- **allow_duplicates** (`Optional[bool]`) – Whether or not column can contain duplicate values.

Warning: This option will be deprecated in 0.8.0. Use the `unique` argument instead.

Parameters

- **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`.
- **name** (`Optional[str]`) – series name.
- **pandas_dtype** (`Union[str, type, DataType, ExtensionDtype, dtype, None]`) – alias of `dtype` for backwards compatibility.

Warning: This option will be deprecated in 0.8.0

7.15.1.1.2.2 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(
...     float, [
...         pa.Check(lambda s: s > 0),
...         pa.Check(lambda s: s < 1000),
...         pa.Check(lambda s: s.mean() > 300),
...     ])
>>> series = pd.Series([1, 100, 800, 900, 999], dtype=float)
>>> print(series_schema.validate(series))
0      1.0
1    100.0
2    800.0
3    900.0
4    999.0
dtype: float64
```

7.15.1.1.2.3 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`

7.15.1.2 Schema Components

<code>pandera.schema_components.Column</code>	Validate types and properties of <code>DataFrame</code> columns.
<code>pandera.schema_components.Index</code>	Validate types and properties of a <code>DataFrame</code> Index.
<code>pandera.schema_components.MultiIndex</code>	Validate types and properties of a <code>DataFrame</code> MultiIndex.

7.15.1.2.1 pandera.schema_components.Column

```
class pandera.schema_components.Column(dtype=None, checks=None, nullable=False, unique=False,
    allow_duplicates=None, coerce=False, required=True,
    name=None, regex=False, pandas_dtype=None)
```

Validate types and properties of DataFrame columns.

Create column validator object.

Parameters

- **dtype** (`Union[str, type, DataType, 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.
- **unique** (`bool`) – whether column values should be unique
- **allow_duplicates** (`Optional[bool]`) – Whether or not column can contain duplicate values.

Warning: This option will be deprecated in 0.8.0. Use the `unique` argument instead.

- **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** (`Union[str, Tuple[str, ...], None]`) – 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.
- **pandas_dtype** (`Union[str, type, DataType, ExtensionDtype, dtype, None]`) – alias of dtype for backwards compatibility.

Warning: This option will be deprecated in 0.8.0

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

Example

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

(continues on next page)

```

column
0    foo
1    bar

```

See [here](#) for more usage details.

Attributes

<code>allow_duplicates</code>	Whether to allow duplicate values.
<code>checks</code>	Return list of checks or hypotheses.
<code>coerce</code>	Whether to coerce series to specified type.
<code>dtype</code>	Get the pandas dtype
<code>name</code>	Get SeriesSchema name.
<code>nullable</code>	Whether the series is nullable.
<code>properties</code>	Get column properties.
<code>regex</code>	True if name attribute should be treated as a regex pattern.
<code>unique</code>	Whether to check for duplicates in check object

Methods

<code>__init__</code>	Create column validator object.
<code>coerce_dtype</code>	Coerce dtype of a column, handling duplicate column names.
<code>example</code>	Generate an example of a particular size.
<code>get_regex_columns</code>	Get matching column names based on regex column name pattern.
<code>set_name</code>	Used to set or modify the name of a column object.
<code>strategy</code>	Create a hypothesis strategy for generating a Column.
<code>strategy_component</code>	Generate column data object for use by DataFrame strategy.
<code>validate</code>	Validate a Column in a DataFrame object.
<code>__call__</code>	Alias for <code>validate</code> method.

7.15.1.2.1.1 `pandera.schema_components.Column.__init__`

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

Create column validator object.

Parameters

- **dtype** (`Union[str, type, DataType, 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.
- **unique** (`bool`) – whether column values should be unique
- **allow_duplicates** (`Optional[bool]`) – Whether or not column can contain duplicate values.

Warning: This option will be deprecated in 0.8.0. Use the `unique` argument instead.

- **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** (`Union[str, Tuple[str, ...], None]`) – 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.
- **pandas_dtype** (`Union[str, type, DataType, ExtensionDtype, dtype, None]`) – alias of `dtype` for backwards compatibility.

Warning: This option will be deprecated in 0.8.0

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

Example

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

See [here](#) for more usage details.

7.15.1.2.1.2 `pandera.schema_components.Column.coerce_dtype`

`Column.coerce_dtype(obj)`

Coerce dtype of a column, handling duplicate column names.

7.15.1.2.1.3 `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.

7.15.1.2.1.4 `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

7.15.1.2.1.5 `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

7.15.1.2.1.6 `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.

7.15.1.2.1.7 `pandera.schema_components.Column.strategy_component`

`Column.strategy_component()`

Generate column data object for use by DataFrame strategy.

7.15.1.2.1.8 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.

7.15.1.2.1.9 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]`

7.15.1.2.2 pandera.schema_components.Index

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

Validate types and properties of a DataFrame Index.

Initialize series schema base object.

Parameters

- **dtype** (`Union[str, type, DataType, 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** (`Union[Check, Hypothesis, List[Union[Check, Hypothesis]], None]`) – 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.
- **unique** (*bool*) – Whether or not column can contain duplicate values.
- **allow_duplicates** (*Optional[bool]*) – Whether or not column can contain duplicate values.

Warning: This option will be deprecated in 0.8.0. Use the `unique` argument instead.

Parameters

- **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 `dtype=None`.
- **name** (*Optional[Any]*) – column name in dataframe to validate.
- **pandas_dtype** (*Union[str, type, DataType, ExtensionDtype, dtype, None]*) – alias of dtype for backwards compatibility.

Warning: This option will be deprecated in 0.8.0

Attributes

<code>allow_duplicates</code>	Whether to allow duplicate values.
<code>checks</code>	Return list of checks or hypotheses.
<code>coerce</code>	Whether to coerce series to specified type.
<code>dtype</code>	Get the pandas dtype
<code>name</code>	Get SeriesSchema name.
<code>names</code>	Get index names in the Index schema component.
<code>nullable</code>	Whether the series is nullable.
<code>unique</code>	Whether to check for duplicates in check object

Methods

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

7.15.1.2.2.1 `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.

7.15.1.2.2.2 `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.

7.15.1.2.2.3 `pandera.schema_components.Index.strategy_component`

`Index.strategy_component()`

Generate column data object for use by MultiIndex strategy.

7.15.1.2.2.4 `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.

7.15.1.2.2.5 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]`

7.15.1.2.3 pandera.schema_components.MultiIndex

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

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 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.
- **unique** (`Union[str, List[str], None]`) – a list of index names that should be jointly unique.

Example

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>> schema = pa.DataFrameSchema(
...     columns={"column": pa.Column(int)},
...     index=pa.MultiIndex([
...         pa.Index(str,
...             pa.Check(lambda s: s.isin(["foo", "bar"]))),
...         pa.Index(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"],
...     )
... )
```

(continues on next page)

(continued from previous page)

```
>>>
>>> schema.validate(df)
           column
index0 index1
foo    0         1
bar    1         2
foo    2         3
```

See [here](#) for more usage details.

Attributes

<code>coerce</code>	Whether or not to coerce data types.
<code>dtype</code>	Get the dtype property.
<code>dtypes</code>	A dict where the keys are column names and values are <i>DataTypes</i> for the column.
<code>names</code>	Get index names in the MultiIndex schema component.
<code>ordered</code>	Whether or not to validate the columns order.
<code>unique</code>	List of columns that should be jointly unique.

Methods

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

7.15.1.2.3.1 pandera.schema_components.MultiIndex.__init__

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

Parameters

- **indexes** (`List[Index]`) – list of Index validators for each level of the MultiIndex index.
- **coerce** (`bool`) – Whether or not to coerce the MultiIndex to the specified 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.

- **unique** (`Union[str, List[str], None]`) – a list of index names that should be jointly unique.

Example

```

>>> import pandas as pd
>>> import pandera as pa
>>>
>>>
>>> schema = pa.DataFrameSchema(
...     columns={"column": pa.Column(int)},
...     index=pa.MultiIndex([
...         pa.Index(str,
...             pa.Check(lambda s: s.isin(["foo", "bar"]))),
...         pa.Index(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)

```

	index0	index1	column
	foo	0	1
	bar	1	2
	foo	2	3

See [here](#) for more usage details.

7.15.1.2.3.2 `pandera.schema_components.MultiIndex.coerce_dtype`

`MultiIndex.coerce_dtype(obj)`

Coerce type of a `pd.Series` by type specified in `dtype`.

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

Return type `MultiIndex`

Returns `MultiIndex` with coerced data type

7.15.1.2.3.3 `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.

7.15.1.2.3.4 `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.

7.15.1.2.3.5 `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 or 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.

7.15.1.2.3.6 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.

7.15.1.3 Checks

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

7.15.1.3.1 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** (`Union[Callable[[Series], Union[Series, bool]], Callable[[DataFrame], Union[DataFrame, Series, bool]]]`) – 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)

```

>>>
>>> # 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(int, checks=measure_checks),
...         "measure_2": pa.Column(int, checks=measure_checks),
...         "group": pa.Column(str),
...     },
...     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

<code>statistics</code>	Get check statistics.
<code>REGISTERED_CUSTOM_CHECKS</code>	

Methods

<code>eq</code>	Ensure all elements of a series equal a certain value.
<code>equal_to</code>	Ensure all elements of a series equal a certain value.
<code>ge</code>	Ensure all values are greater or equal a certain value.
<code>greater_than</code>	Ensure values of a series are strictly greater than a minimum value.
<code>greater_than_or_equal_to</code>	Ensure all values are greater or equal a certain value.
<code>gt</code>	Ensure values of a series are strictly greater than a minimum value.
<code>in_range</code>	Ensure all values of a series are within an interval.

continues on next page

Table 15 – continued from previous page

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

7.15.1.3.1.1 `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

7.15.1.3.1.2 `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

7.15.1.3.1.3 `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

7.15.1.3.1.4 `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

7.15.1.3.1.5 `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

7.15.1.3.1.6 `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

7.15.1.3.1.7 `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 `max_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

7.15.1.3.1.8 `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()`.

7.15.1.3.1.9 `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

7.15.1.3.1.10 `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

7.15.1.3.1.11 `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

7.15.1.3.1.12 `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

7.15.1.3.1.13 `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

7.15.1.3.1.14 `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

7.15.1.3.1.15 `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 parameter for `forbidden_values` is understood as set of prohibited characters. Any string of length > 1 can't be in it by design.

7.15.1.3.1.16 `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()`.

7.15.1.3.1.17 `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

7.15.1.3.1.18 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

7.15.1.3.1.19 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()`.

7.15.1.3.1.20 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

7.15.1.3.1.21 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 or 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.

7.15.1.3.2 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* assesses statistical significance, and ***kwargs* are other arguments supplied via the ***relationship_kwargs* argument.

Default is “equal” for the null hypothesis.

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

Examples

Define a two-sample hypothesis test using scipy.

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>> from scipy import stats
>>>
>>> schema = pa.DataFrameSchema({
...     "height_in_feet": pa.Column(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(str),
... })
>>> 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.
REGISTERED_CUSTOM_CHECKS	

Methods

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

7.15.1.3.2.1 `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` assesses

statistical significance, and ***kwargs* are other arguments supplied via the ***relationship_kwargs* argument.

Default is “equal” for the null hypothesis.

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

Examples

Define a two-sample hypothesis test using scipy.

```
>>> import pandas as pd
>>> import pandera as pa
>>>
>>> from scipy import stats
>>>
>>> schema = pa.DataFrameSchema({
...     "height_in_feet": pa.Column(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(str),
... })
>>> 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.

7.15.1.3.2.2 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(
...         float, [
...             pa.Hypothesis.one_sample_ttest(
...                 popmean=5,
...                 relationship="greater_than",
...                 alpha=0.1),
...         ]),
... })
>>> df = (
...     pd.DataFrame({
```

(continues on next page)

(continued from previous page)

```

...     "height_in_feet": [8.1, 7, 6.5, 6.7, 5.1],
...     })
... )
>>> schema.validate(df)
   height_in_feet
0              8.1
1              7.0
2              6.5
3              6.7
4              5.1

```

7.15.1.3.2.3 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(
...         float, [
...             pa.Hypothesis.two_sample_ttest(
...                 sample1="A",
...                 sample2="B",
...                 groupby="group",
...                 relationship="greater_than",
...                 alpha=0.05,
...                 equal_var=True),
...         ]),
...     "group": pa.Column(str)
... })
>>> 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
```

7.15.1.3.2.4 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.

7.15.2 Data Types

7.15.2.1 Library-agnostic dtypes

<code>pandera.dtypes.DataType</code>	Base class of all Pandera data types.
<code>pandera.dtypes.Bool</code>	Semantic representation of a boolean data type.
<code>pandera.dtypes.Timestamp</code>	Semantic representation of a timestamp data type.
<code>pandera.dtypes.DateTime</code>	alias of <code>pandera.dtypes.Timestamp</code>
<code>pandera.dtypes.Timedelta</code>	Semantic representation of a delta time data type.
<code>pandera.dtypes.Category</code>	Semantic representation of a categorical data type.
<code>pandera.dtypes.Float</code>	Semantic representation of a floating data type.
<code>pandera.dtypes.Float16</code>	Semantic representation of a floating data type stored in 16 bits.
<code>pandera.dtypes.Float32</code>	Semantic representation of a floating data type stored in 32 bits.
<code>pandera.dtypes.Float64</code>	Semantic representation of a floating data type stored in 64 bits.
<code>pandera.dtypes.Float128</code>	Semantic representation of a floating data type stored in 128 bits.
<code>pandera.dtypes.Int</code>	Semantic representation of an integer data type.
<code>pandera.dtypes.Int8</code>	Semantic representation of an integer data type stored in 8 bits.
<code>pandera.dtypes.Int16</code>	Semantic representation of an integer data type stored in 16 bits.
<code>pandera.dtypes.Int32</code>	Semantic representation of an integer data type stored in 32 bits.
<code>pandera.dtypes.Int64</code>	Semantic representation of an integer data type stored in 64 bits.
<code>pandera.dtypes.UInt</code>	Semantic representation of an unsigned integer data type.
<code>pandera.dtypes.UInt8</code>	Semantic representation of an unsigned integer data type stored in 8 bits.
<code>pandera.dtypes.UInt16</code>	Semantic representation of an unsigned integer data type stored in 16 bits.
<code>pandera.dtypes.UInt32</code>	Semantic representation of an unsigned integer data type stored in 32 bits.
<code>pandera.dtypes.UInt64</code>	Semantic representation of an unsigned integer data type stored in 64 bits.
<code>pandera.dtypes.Complex</code>	Semantic representation of a complex number data type.
<code>pandera.dtypes.Complex64</code>	Semantic representation of a complex number data type stored in 64 bits.

continues on next page

Table 18 – continued from previous page

<code>pandera.dtypes.Complex128</code>	Semantic representation of a complex number data type stored in 128 bits.
<code>pandera.dtypes.Complex256</code>	Semantic representation of a complex number data type stored in 256 bits.
<code>pandera.dtypes.String</code>	Semantic representation of a string data type.

7.15.2.1.1 `pandera.dtypes.DataType`

class `pandera.dtypes.DataType`

Base class of all Pandera data types.

Attributes

<code>continuous</code>	Whether the number data type is continuous.
-------------------------	---

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.1.1 `pandera.dtypes.DataType.__init__`

`DataType.__init__()`

7.15.2.1.1.2 `pandera.dtypes.DataType.check`

`DataType.check(pandera_dtype)`

Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.1.3 `pandera.dtypes.DataType.coerce`

`DataType.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.1.4 `pandera.dtypes.DataType.try_coerce`

`DataType.try_coerce(data_container)`
Coerce data container to the data type, raises a `~pandera.errors.ParserError` if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.1.5 `pandera.dtypes.DataType.__call__`

`DataType.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.2 `pandera.dtypes.Bool`

class `pandera.dtypes.Bool`
Semantic representation of a boolean data type.

Attributes

<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.

Methods

<code>__init__</code>	
<code>check</code>	Check that <code>pandera.DataType</code> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <code>~pandera.errors.ParserError</code> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.2.1 `pandera.dtypes.Bool.__init__`

`Bool.__init__()`

7.15.2.1.2.2 `pandera.dtypes.Bool.check`

`Bool.check(pandera_dtype)`
 Check that pandera *DataType* are equivalent.
Return type `bool`

7.15.2.1.2.3 `pandera.dtypes.Bool.coerce`

`Bool.coerce(data_container)`
 Coerce data container to the data type.

7.15.2.1.2.4 `pandera.dtypes.Bool.try_coerce`

`Bool.try_coerce(data_container)`
 Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails
Raises `ParserError`: if coercion fails

7.15.2.1.2.5 `pandera.dtypes.Bool.__call__`

`Bool.__call__(data_container)`
 Coerce data container to the data type.

7.15.2.1.3 `pandera.dtypes.Timestamp`

class `pandera.dtypes.Timestamp`
 Semantic representation of a timestamp data type.

Attributes

<code>continuous</code>	Whether the number data type is continuous.
-------------------------	---

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails

continues on next page

Table 24 – continued from previous page

<code>__call__</code>	Coerce data container to the data type.
-----------------------	---

7.15.2.1.3.1 `pandera.dtypes.Timestamp.__init__`

`Timestamp.__init__()`

7.15.2.1.3.2 `pandera.dtypes.Timestamp.check`

`Timestamp.check(pandera_dtype)`

Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.3.3 `pandera.dtypes.Timestamp.coerce`

`Timestamp.coerce(data_container)`

Coerce data container to the data type.

7.15.2.1.3.4 `pandera.dtypes.Timestamp.try_coerce`

`Timestamp.try_coerce(data_container)`

Coerce data container to the data type, raises a `~pandera.errors.ParserError` if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.3.5 `pandera.dtypes.Timestamp.__call__`

`Timestamp.__call__(data_container)`

Coerce data container to the data type.

7.15.2.1.4 `pandera.dtypes.DateTime`

`pandera.dtypes.DateTime`

alias of `pandera.dtypes.Timestamp`

7.15.2.1.5 `pandera.dtypes.Timedelta`

`class pandera.dtypes.Timedelta`

Semantic representation of a delta time data type.

Attributes

continuous	Whether the number data type is continuous.
------------	---

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.5.1 pandera.dtypes.Timedelta.__init__

`Timedelta.__init__()`

7.15.2.1.5.2 pandera.dtypes.Timedelta.check

`Timedelta.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.5.3 pandera.dtypes.Timedelta.coerce

`Timedelta.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.5.4 pandera.dtypes.Timedelta.try_coerce

`Timedelta.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.5.5 pandera.dtypes.Timedelta.__call__

`Timedelta.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.6 pandera.dtypes.Category

class `pandera.dtypes.Category`(*categories=None, ordered=False*)
 Semantic representation of a categorical data type.

Attributes

<code>categories</code>	
<code>continuous</code>	Whether the number data type is continuous.
<code>ordered</code>	

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.6.1 pandera.dtypes.Category.__init__

`Category.__init__`(*categories=None, ordered=False*)

7.15.2.1.6.2 pandera.dtypes.Category.check

`Category.check`(*pandera_dtype*)
 Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.6.3 pandera.dtypes.Category.coerce

`Category.coerce`(*data_container*)
 Coerce data container to the data type.

7.15.2.1.6.4 `pandera.dtypes.Category.try_coerce`

`Category.try_coerce(data_container)`

Coerce data container to the data type, raises a `~pandera.errors.ParserError` if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.6.5 `pandera.dtypes.Category.__call__`

`Category.__call__(data_container)`

Coerce data container to the data type.

7.15.2.1.7 `pandera.dtypes.Float`

class `pandera.dtypes.Float`

Semantic representation of a floating data type.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.

Methods

<code>__init__</code>	
<code>check</code>	Check that <code>pandera <i>DataType</i></code> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <code>~pandera.errors.ParserError</code> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.7.1 `pandera.dtypes.Float.__init__`

`Float.__init__()`

7.15.2.1.7.2 pandera.dtypes.Float.check

`Float.check(pandera_dtype)`

Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.7.3 pandera.dtypes.Float.coerce

`Float.coerce(data_container)`

Coerce data container to the data type.

7.15.2.1.7.4 pandera.dtypes.Float.try_coerce

`Float.try_coerce(data_container)`

Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.7.5 pandera.dtypes.Float.__call__

`Float.__call__(data_container)`

Coerce data container to the data type.

7.15.2.1.8 pandera.dtypes.Float16

class `pandera.dtypes.Float16`

Semantic representation of a floating data type stored in 16 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.8.1 `pandera.dtypes.Float16.__init__`

`Float16.__init__()`

7.15.2.1.8.2 `pandera.dtypes.Float16.check`

`Float16.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.
Return type `bool`

7.15.2.1.8.3 `pandera.dtypes.Float16.coerce`

`Float16.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.8.4 `pandera.dtypes.Float16.try_coerce`

`Float16.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails
Raises `ParserError`: if coercion fails

7.15.2.1.8.5 `pandera.dtypes.Float16.__call__`

`Float16.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.9 `pandera.dtypes.Float32`

class `pandera.dtypes.Float32`
Semantic representation of a floating data type stored in 32 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.9.1 pandera.dtypes.Float32.__init__

`Float32.__init__()`

7.15.2.1.9.2 pandera.dtypes.Float32.check

`Float32.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.9.3 pandera.dtypes.Float32.coerce

`Float32.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.9.4 pandera.dtypes.Float32.try_coerce

`Float32.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.9.5 pandera.dtypes.Float32.__call__

`Float32.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.10 pandera.dtypes.Float64

class `pandera.dtypes.Float64`

Semantic representation of a floating data type stored in 64 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.10.1 pandera.dtypes.Float64.__init__

`Float64.__init__()`

7.15.2.1.10.2 pandera.dtypes.Float64.check

`Float64.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.10.3 pandera.dtypes.Float64.coerce

`Float64.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.10.4 pandera.dtypes.Float64.try_coerce

`Float64.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.10.5 `pandera.dtypes.Float64.__call__`

`Float64.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.11 `pandera.dtypes.Float128`

class `pandera.dtypes.Float128`

Semantic representation of a floating data type stored in 128 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.11.1 `pandera.dtypes.Float128.__init__`

`Float128.__init__()`

7.15.2.1.11.2 `pandera.dtypes.Float128.check`

`Float128.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.11.3 pandera.dtypes.Float128.coerce

`Float128.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.11.4 pandera.dtypes.Float128.try_coerce

`Float128.try_coerce(data_container)`
Coerce data container to the data type, raises a *~pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.11.5 pandera.dtypes.Float128.__call__

`Float128.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.12 pandera.dtypes.Int

class `pandera.dtypes.Int`
Semantic representation of an integer data type.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>~pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.12.1 pandera.dtypes.Int.__init__

`Int.__init__()`

7.15.2.1.12.2 pandera.dtypes.Int.check

`Int.check(pandera_dtype)`

Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.12.3 pandera.dtypes.Int.coerce

`Int.coerce(data_container)`

Coerce data container to the data type.

7.15.2.1.12.4 pandera.dtypes.Int.try_coerce

`Int.try_coerce(data_container)`

Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.12.5 pandera.dtypes.Int.__call__

`Int.__call__(data_container)`

Coerce data container to the data type.

7.15.2.1.13 pandera.dtypes.Int8

class `pandera.dtypes.Int8`

Semantic representation of an integer data type stored in 8 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.13.1 `pandera.dtypes.Int8.__init__`

`Int8.__init__()`

7.15.2.1.13.2 `pandera.dtypes.Int8.check`

`Int8.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.13.3 `pandera.dtypes.Int8.coerce`

`Int8.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.13.4 `pandera.dtypes.Int8.try_coerce`

`Int8.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.13.5 `pandera.dtypes.Int8.__call__`

`Int8.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.14 `pandera.dtypes.Int16`

class `pandera.dtypes.Int16`
Semantic representation of an integer data type stored in 16 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.14.1 pandera.dtypes.Int16.__init__

`Int16.__init__()`

7.15.2.1.14.2 pandera.dtypes.Int16.check

`Int16.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.14.3 pandera.dtypes.Int16.coerce

`Int16.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.14.4 pandera.dtypes.Int16.try_coerce

`Int16.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.14.5 pandera.dtypes.Int16.__call__

`Int16.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.15 pandera.dtypes.Int32

class `pandera.dtypes.Int32`
Semantic representation of an integer data type stored in 32 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.15.1 pandera.dtypes.Int32.__init__

`Int32.__init__()`

7.15.2.1.15.2 pandera.dtypes.Int32.check

`Int32.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.15.3 `pandera.dtypes.Int32.coerce`

`Int32.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.15.4 `pandera.dtypes.Int32.try_coerce`

`Int32.try_coerce(data_container)`
Coerce data container to the data type, raises a `~pandera.errors.ParserError` if the coercion fails
Raises `ParserError`: if coercion fails

7.15.2.1.15.5 `pandera.dtypes.Int32.__call__`

`Int32.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.16 `pandera.dtypes.Int64`

class `pandera.dtypes.Int64`
Semantic representation of an integer data type stored in 64 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.

Methods

<code>__init__</code>	
<code>check</code>	Check that <code>pandera <i>DataType</i></code> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <code>~pandera.errors.ParserError</code> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.16.1 `pandera.dtypes.Int64.__init__`

`Int64.__init__()`

7.15.2.1.16.2 `pandera.dtypes.Int64.check`

`Int64.check(pandera_dtype)`

Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.16.3 `pandera.dtypes.Int64.coerce`

`Int64.coerce(data_container)`

Coerce data container to the data type.

7.15.2.1.16.4 `pandera.dtypes.Int64.try_coerce`

`Int64.try_coerce(data_container)`

Coerce data container to the data type, raises a *~pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.16.5 `pandera.dtypes.Int64.__call__`

`Int64.__call__(data_container)`

Coerce data container to the data type.

7.15.2.1.17 `pandera.dtypes.UInt`

class `pandera.dtypes.UInt`

Semantic representation of an unsigned integer data type.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.17.1 pandera.dtypes.UInt.__init__

`UInt.__init__()`

7.15.2.1.17.2 pandera.dtypes.UInt.check

`UInt.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.17.3 pandera.dtypes.UInt.coerce

`UInt.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.17.4 pandera.dtypes.UInt.try_coerce

`UInt.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.17.5 pandera.dtypes.UInt.__call__

`UInt.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.18 pandera.dtypes.UInt8

class `pandera.dtypes.UInt8`

Semantic representation of an unsigned integer data type stored in 8 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.18.1 `pandera.dtypes.UInt8.__init__`

`UInt8.__init__()`

7.15.2.1.18.2 `pandera.dtypes.UInt8.check`

`UInt8.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.18.3 `pandera.dtypes.UInt8.coerce`

`UInt8.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.18.4 `pandera.dtypes.UInt8.try_coerce`

`UInt8.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.18.5 pandera.dtypes.UInt8.__call__

`UInt8.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.19 pandera.dtypes.UInt16

class `pandera.dtypes.UInt16`

Semantic representation of an unsigned integer data type stored in 16 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.19.1 pandera.dtypes.UInt16.__init__

`UInt16.__init__()`

7.15.2.1.19.2 pandera.dtypes.UInt16.check

`UInt16.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.19.3 pandera.dtypes.UInt16.coerce

`UInt16.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.19.4 pandera.dtypes.UInt16.try_coerce

`UInt16.try_coerce(data_container)`
Coerce data container to the data type, raises a `~pandera.errors.ParserError` if the coercion fails
Raises `ParserError`: if coercion fails

7.15.2.1.19.5 pandera.dtypes.UInt16.__call__

`UInt16.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.20 pandera.dtypes.UInt32

class `pandera.dtypes.UInt32`
Semantic representation of an unsigned integer data type stored in 32 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <code>DataType</code> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <code>~pandera.errors.ParserError</code> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.20.1 `pandera.dtypes.UInt32.__init__`

`UInt32.__init__()`

7.15.2.1.20.2 `pandera.dtypes.UInt32.check`

`UInt32.check(pandera_dtype)`

Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.20.3 `pandera.dtypes.UInt32.coerce`

`UInt32.coerce(data_container)`

Coerce data container to the data type.

7.15.2.1.20.4 `pandera.dtypes.UInt32.try_coerce`

`UInt32.try_coerce(data_container)`

Coerce data container to the data type, raises a *~pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.20.5 `pandera.dtypes.UInt32.__call__`

`UInt32.__call__(data_container)`

Coerce data container to the data type.

7.15.2.1.21 `pandera.dtypes.UInt64`

class `pandera.dtypes.UInt64`

Semantic representation of an unsigned integer data type stored in 64 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.21.1 pandera.dtypes.UInt64.__init__

`UInt64.__init__()`

7.15.2.1.21.2 pandera.dtypes.UInt64.check

`UInt64.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.21.3 pandera.dtypes.UInt64.coerce

`UInt64.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.21.4 pandera.dtypes.UInt64.try_coerce

`UInt64.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.21.5 pandera.dtypes.UInt64.__call__

`UInt64.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.22 pandera.dtypes.Complex

class `pandera.dtypes.Complex`
Semantic representation of a complex number data type.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.22.1 pandera.dtypes.Complex.__init__

`Complex.__init__()`

7.15.2.1.22.2 pandera.dtypes.Complex.check

`Complex.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.22.3 pandera.dtypes.Complex.coerce

`Complex.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.22.4 pandera.dtypes.Complex.try_coerce

`Complex.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.22.5 `pandera.dtypes.Complex.__call__`

`Complex.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.23 `pandera.dtypes.Complex64`

class `pandera.dtypes.Complex64`

Semantic representation of a complex number data type stored in 64 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.23.1 `pandera.dtypes.Complex64.__init__`

`Complex64.__init__()`

7.15.2.1.23.2 `pandera.dtypes.Complex64.check`

`Complex64.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.23.3 `pandera.dtypes.Complex64.coerce`

`Complex64.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.23.4 `pandera.dtypes.Complex64.try_coerce`

`Complex64.try_coerce(data_container)`
Coerce data container to the data type, raises a `~pandera.errors.ParserError` if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.23.5 `pandera.dtypes.Complex64.__call__`

`Complex64.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.24 `pandera.dtypes.Complex128`

class `pandera.dtypes.Complex128`

Semantic representation of a complex number data type stored in 128 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.

Methods

<code>__init__</code>	
<code>check</code>	Check that <code>pandera.DataType</code> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <code>~pandera.errors.ParserError</code> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.24.1 `pandera.dtypes.Complex128.__init__`

`Complex128.__init__()`

7.15.2.1.24.2 `pandera.dtypes.Complex128.check`

`Complex128.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.24.3 `pandera.dtypes.Complex128.coerce`

`Complex128.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.24.4 `pandera.dtypes.Complex128.try_coerce`

`Complex128.try_coerce(data_container)`
Coerce data container to the data type, raises a `~pandera.errors.ParserError` if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.24.5 `pandera.dtypes.Complex128.__call__`

`Complex128.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.25 `pandera.dtypes.Complex256`

class `pandera.dtypes.Complex256`
Semantic representation of a complex number data type stored in 256 bits.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.25.1 pandera.dtypes.Complex256.__init__

`Complex256.__init__()`

7.15.2.1.25.2 pandera.dtypes.Complex256.check

`Complex256.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.25.3 pandera.dtypes.Complex256.coerce

`Complex256.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.25.4 pandera.dtypes.Complex256.try_coerce

`Complex256.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.25.5 pandera.dtypes.Complex256.__call__

`Complex256.__call__(data_container)`
Coerce data container to the data type.

7.15.2.1.26 pandera.dtypes.String

class `pandera.dtypes.String`
Semantic representation of a string data type.

Attributes

continuous	Whether the number data type is continuous.
------------	---

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Coerce data container to the data type.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.1.26.1 pandera.dtypes.String.__init__

`String.__init__()`

7.15.2.1.26.2 pandera.dtypes.String.check

`String.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.1.26.3 pandera.dtypes.String.coerce

`String.coerce(data_container)`
Coerce data container to the data type.

7.15.2.1.26.4 pandera.dtypes.String.try_coerce

`String.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

7.15.2.1.26.5 pandera.dtypes.String.__call__

`String.__call__(data_container)`
Coerce data container to the data type.

7.15.2.2 Pandas-specific Dtypes

Listed here for compatibility with pandera versions < 0.7. Passing native pandas dtypes to pandera components is preferred.

<code>pandera.engines.pandas_engine.BOOL</code>	Semantic representation of a <code>pandas.BooleanDtype</code> .
<code>pandera.engines.pandas_engine.INT8</code>	Semantic representation of a <code>pandas.Int8Dtype</code> .
<code>pandera.engines.pandas_engine.INT16</code>	Semantic representation of a <code>pandas.Int16Dtype</code> .
<code>pandera.engines.pandas_engine.INT32</code>	Semantic representation of a <code>pandas.Int32Dtype</code> .
<code>pandera.engines.pandas_engine.INT64</code>	Semantic representation of a <code>pandas.Int64Dtype</code> .
<code>pandera.engines.pandas_engine.UINT8</code>	Semantic representation of a <code>pandas.UInt8Dtype</code> .
<code>pandera.engines.pandas_engine.UINT16</code>	Semantic representation of a <code>pandas.UInt16Dtype</code> .
<code>pandera.engines.pandas_engine.UINT32</code>	Semantic representation of a <code>pandas.UInt32Dtype</code> .
<code>pandera.engines.pandas_engine.UINT64</code>	Semantic representation of a <code>pandas.UInt64Dtype</code> .
<code>pandera.engines.pandas_engine.STRING</code>	Semantic representation of a <code>pandas.StringDtype</code> .
<code>pandera.engines.numpy_engine.Object</code>	Semantic representation of a <code>numpy.object_</code> .

7.15.2.2.1 pandera.engines.pandas_engine.BOOL

class `pandera.engines.pandas_engine.BOOL`
 Semantic representation of a `pandas.BooleanDtype`.

Attributes

<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>type</code>	Native pandas dtype boxed by the data type.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <code>DataTypes</code> are equivalent.
<code>coerce</code>	Pure coerce without catching exceptions.
<code>try_coerce</code>	Coerce data container to the data type, raises a <code>pandera.errors.ParserError</code> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.2.1.1 pandera.engines.pandas_engine.BOOL.__init__

`BOOL.__init__()`

7.15.2.2.1.2 pandera.engines.pandas_engine.BOOL.check

`BOOL.check(pandera_dtype)`

Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.2.1.3 pandera.engines.pandas_engine.BOOL.coerce

`BOOL.coerce(data_container)`

Pure coerce without catching exceptions.

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.1.4 pandera.engines.pandas_engine.BOOL.try_coerce

`BOOL.try_coerce(data_container)`

Coerce data container to the data type, raises a *~pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.1.5 pandera.engines.pandas_engine.BOOL.__call__

`BOOL.__call__(data_container)`

Coerce data container to the data type.

7.15.2.2.2 pandera.engines.pandas_engine.INT8

class `pandera.engines.pandas_engine.INT8`

Semantic representation of a `pandas.Int8Dtype`.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.
<code>type</code>	Native pandas dtype boxed by the data type.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Pure coerce without catching exceptions.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.2.2.1 pandera.engines.pandas_engine.INT8.__init__

INT8.__init__()

7.15.2.2.2.2 pandera.engines.pandas_engine.INT8.check

INT8.**check**(*pandera_dtype*)
Check that pandera *DataType* are equivalent.

Return type bool

7.15.2.2.2.3 pandera.engines.pandas_engine.INT8.coerce

INT8.**coerce**(*data_container*)
Pure coerce without catching exceptions.

Return type Union[Series, Index, DataFrame]

7.15.2.2.2.4 pandera.engines.pandas_engine.INT8.try_coerce

INT8.**try_coerce**(*data_container*)
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises ParserError: if coercion fails

Return type Union[Series, Index, DataFrame]

7.15.2.2.2.5 pandera.engines.pandas_engine.INT8.__call__

INT8.**__call__**(*data_container*)
Coerce data container to the data type.

7.15.2.2.3 pandera.engines.pandas_engine.INT16

class `pandera.engines.pandas_engine.INT16`
 Semantic representation of a `pandas.Int16Dtype`.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.
<code>type</code>	Native pandas dtype boxed by the data type.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Pure coerce without catching exceptions.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.2.3.1 pandera.engines.pandas_engine.INT16.__init__

`INT16.__init__()`

7.15.2.2.3.2 pandera.engines.pandas_engine.INT16.check

`INT16.check(pandera_dtype)`
 Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.2.3.3 pandera.engines.pandas_engine.INT16.coerce

`INT16.coerce(data_container)`
 Pure coerce without catching exceptions.

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.3.4 pandera.engines.pandas_engine.INT16.try_coerce

INT16.**try_coerce**(*data_container*)

Coerce data container to the data type, raises a *~pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.3.5 pandera.engines.pandas_engine.INT16.__call__

INT16.**__call__**(*data_container*)

Coerce data container to the data type.

7.15.2.2.4 pandera.engines.pandas_engine.INT32

class `pandera.engines.pandas_engine.INT32`

Semantic representation of a `pandas.Int32Dtype`.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.
<code>type</code>	Native pandas dtype boxed by the data type.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Pure coerce without catching exceptions.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>~pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.2.4.1 pandera.engines.pandas_engine.INT32.__init__

`INT32.__init__()`

7.15.2.2.4.2 pandera.engines.pandas_engine.INT32.check

`INT32.check(pandera_dtype)`

Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.2.4.3 pandera.engines.pandas_engine.INT32.coerce

`INT32.coerce(data_container)`

Pure coerce without catching exceptions.

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.4.4 pandera.engines.pandas_engine.INT32.try_coerce

`INT32.try_coerce(data_container)`

Coerce data container to the data type, raises a *~pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.4.5 pandera.engines.pandas_engine.INT32.__call__

`INT32.__call__(data_container)`

Coerce data container to the data type.

7.15.2.2.5 pandera.engines.pandas_engine.INT64

class `pandera.engines.pandas_engine.INT64`

Semantic representation of a `pandas.Int64Dtype`.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.
<code>type</code>	Native pandas dtype boxed by the data type.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Pure coerce without catching exceptions.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.2.5.1 pandera.engines.pandas_engine.INT64.__init__

INT64.`__init__`()

7.15.2.2.5.2 pandera.engines.pandas_engine.INT64.check

INT64.`check`(*pandera_dtype*)

Check that pandera *DataType* are equivalent.

Return type bool

7.15.2.2.5.3 pandera.engines.pandas_engine.INT64.coerce

INT64.`coerce`(*data_container*)

Pure coerce without catching exceptions.

Return type Union[Series, Index, DataFrame]

7.15.2.2.5.4 pandera.engines.pandas_engine.INT64.try_coerce

INT64.`try_coerce`(*data_container*)

Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises ParserError: if coercion fails

Return type Union[Series, Index, DataFrame]

7.15.2.2.5.5 pandera.engines.pandas_engine.INT64.__call__

INT64.`__call__`(*data_container*)

Coerce data container to the data type.

7.15.2.2.6 pandera.engines.pandas_engine.UINT8

class `pandera.engines.pandas_engine.UINT8`
 Semantic representation of a `pandas.UInt8Dtype`.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.
<code>type</code>	Native pandas dtype boxed by the data type.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Pure coerce without catching exceptions.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.2.6.1 pandera.engines.pandas_engine.UINT8.__init__

`UINT8.__init__()`

7.15.2.2.6.2 pandera.engines.pandas_engine.UINT8.check

`UINT8.check(pandera_dtype)`
 Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.2.6.3 pandera.engines.pandas_engine.UINT8.coerce

`UINT8.coerce(data_container)`
 Pure coerce without catching exceptions.

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.6.4 pandera.engines.pandas_engine.UINT8.try_coerce

`UINT8.try_coerce(data_container)`

Coerce data container to the data type, raises a `~pandera.errors.ParserError` if the coercion fails

Raises `ParserError`: if coercion fails

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.6.5 pandera.engines.pandas_engine.UINT8.__call__

`UINT8.__call__(data_container)`

Coerce data container to the data type.

7.15.2.2.7 pandera.engines.pandas_engine.UINT16

class `pandera.engines.pandas_engine.UINT16`

Semantic representation of a `pandas.UInt16Dtype`.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.
<code>type</code>	Native pandas dtype boxed by the data type.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <code>DataType</code> are equivalent.
<code>coerce</code>	Pure coerce without catching exceptions.
<code>try_coerce</code>	Coerce data container to the data type, raises a <code>~pandera.errors.ParserError</code> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.2.7.1 pandera.engines.pandas_engine.UINT16.__init__

UINT16.__init__()

7.15.2.2.7.2 pandera.engines.pandas_engine.UINT16.check

UINT16.check(*pandera_dtype*)

Check that pandera *DataType* are equivalent.

Return type bool

7.15.2.2.7.3 pandera.engines.pandas_engine.UINT16.coerce

UINT16.coerce(*data_container*)

Pure coerce without catching exceptions.

Return type Union[Series, Index, DataFrame]

7.15.2.2.7.4 pandera.engines.pandas_engine.UINT16.try_coerce

UINT16.try_coerce(*data_container*)

Coerce data container to the data type, raises a *~pandera.errors.ParserError* if the coercion fails

Raises ParserError: if coercion fails

Return type Union[Series, Index, DataFrame]

7.15.2.2.7.5 pandera.engines.pandas_engine.UINT16.__call__

UINT16.__call__(*data_container*)

Coerce data container to the data type.

7.15.2.2.8 pandera.engines.pandas_engine.UINT32

class pandera.engines.pandas_engine.UINT32

Semantic representation of a pandas.UInt32Dtype.

Attributes

bit_width	Number of bits used by the machine representation.
continuous	Whether the number data type is continuous.
exact	Whether the data type is an exact representation of a number.
signed	Whether the integer data type is signed.
type	Native pandas dtype boxed by the data type.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Pure coerce without catching exceptions.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.2.8.1 pandera.engines.pandas_engine.UINT32.__init__

`UINT32.__init__()`

7.15.2.2.8.2 pandera.engines.pandas_engine.UINT32.check

`UINT32.check(pandera_dtype)`
 Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.2.8.3 pandera.engines.pandas_engine.UINT32.coerce

`UINT32.coerce(data_container)`
 Pure coerce without catching exceptions.

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.8.4 pandera.engines.pandas_engine.UINT32.try_coerce

`UINT32.try_coerce(data_container)`
 Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.8.5 pandera.engines.pandas_engine.UINT32.__call__

`UINT32.__call__(data_container)`
 Coerce data container to the data type.

7.15.2.2.9 pandera.engines.pandas_engine.UINT64

class `pandera.engines.pandas_engine.UINT64`
 Semantic representation of a `pandas.UInt64Dtype`.

Attributes

<code>bit_width</code>	Number of bits used by the machine representation.
<code>continuous</code>	Whether the number data type is continuous.
<code>exact</code>	Whether the data type is an exact representation of a number.
<code>signed</code>	Whether the integer data type is signed.
<code>type</code>	Native pandas dtype boxed by the data type.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Pure coerce without catching exceptions.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.2.9.1 pandera.engines.pandas_engine.UINT64.__init__

`UINT64.__init__()`

7.15.2.2.9.2 pandera.engines.pandas_engine.UINT64.check

`UINT64.check(pandera_dtype)`
 Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.2.9.3 pandera.engines.pandas_engine.UINT64.coerce

`UINT64.coerce(data_container)`
 Pure coerce without catching exceptions.

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.9.4 `pandera.engines.pandas_engine.UINT64.try_coerce`

`UINT64.try_coerce(data_container)`

Coerce data container to the data type, raises a `~pandera.errors.ParserError` if the coercion fails

Raises `ParserError`: if coercion fails

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.9.5 `pandera.engines.pandas_engine.UINT64.__call__`

`UINT64.__call__(data_container)`

Coerce data container to the data type.

7.15.2.2.10 `pandera.engines.pandas_engine.STRING`

`class pandera.engines.pandas_engine.STRING(storage='python')`

Semantic representation of a `pandas.StringDtype`.

Attributes

<code>continuous</code>	Whether the number data type is continuous.
<code>storage</code>	
<code>type</code>	Native pandas dtype boxed by the data type.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <code>DataType</code> are equivalent.
<code>coerce</code>	Pure coerce without catching exceptions.
<code>from_parametrized_dtype</code>	Convert a <code>pandas.StringDtype</code> to a Pandera <code>pandera.engines.pandas_engine.STRING</code> .
<code>try_coerce</code>	Coerce data container to the data type, raises a <code>~pandera.errors.ParserError</code> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.2.10.1 `pandera.engines.pandas_engine.STRING.__init__`

`STRING.__init__(storage='python')`

7.15.2.2.10.2 `pandera.engines.pandas_engine.STRING.check`

`STRING.check(pandera_dtype)`

Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.2.10.3 `pandera.engines.pandas_engine.STRING.coerce`

`STRING.coerce(data_container)`

Pure coerce without catching exceptions.

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.10.4 `pandera.engines.pandas_engine.STRING.from_parametrized_dtype`

`classmethod STRING.from_parametrized_dtype(pd_dtype)`

Convert a `pandas.StringDtype` to a Pandera `pandera.engines.pandas_engine.STRING`.

7.15.2.2.10.5 `pandera.engines.pandas_engine.STRING.try_coerce`

`STRING.try_coerce(data_container)`

Coerce data container to the data type, raises a `~pandera.errors.ParserError` if the coercion fails

Raises `ParserError`: if coercion fails

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.10.6 `pandera.engines.pandas_engine.STRING.__call__`

`STRING.__call__(data_container)`

Coerce data container to the data type.

7.15.2.2.11 `pandera.engines.numpy_engine.Object`

`class pandera.engines.numpy_engine.Object`

Semantic representation of a `numpy.object_`.

Attributes

<code>continuous</code>	Whether the number data type is continuous.
<code>type</code>	Native numpy dtype boxed by the data type.

Methods

<code>__init__</code>	
<code>check</code>	Check that pandera <i>DataType</i> are equivalent.
<code>coerce</code>	Pure coerce without catching exceptions.
<code>try_coerce</code>	Coerce data container to the data type, raises a <i>pandera.errors.ParserError</i> if the coercion fails
<code>__call__</code>	Coerce data container to the data type.

7.15.2.2.11.1 pandera.engines.numpy_engine.Object.__init__

`Object.__init__()`

7.15.2.2.11.2 pandera.engines.numpy_engine.Object.check

`Object.check(pandera_dtype)`
Check that pandera *DataType* are equivalent.

Return type `bool`

7.15.2.2.11.3 pandera.engines.numpy_engine.Object.coerce

`Object.coerce(data_container)`
Pure coerce without catching exceptions.

Return type `Union[Series, Index, DataFrame]`

7.15.2.2.11.4 pandera.engines.numpy_engine.Object.try_coerce

`Object.try_coerce(data_container)`
Coerce data container to the data type, raises a *pandera.errors.ParserError* if the coercion fails

Raises `ParserError`: if coercion fails

Return type `Union[Series, Index, DataFrame, ndarray]`

7.15.2.2.11.5 `pandera.engines.numpy_engine.Object.__call__`

`Object.__call__(data_container)`
Coerce data container to the data type.

7.15.2.3 Utility functions

<code>pandera.dtypes.is_subdtype</code>	Returns True if first argument is lower/equal in <code>DataType</code> hierarchy.
<code>pandera.dtypes.is_float</code>	Return True if <code>pandera.dtypes.DataType</code> is a float.
<code>pandera.dtypes.is_int</code>	Return True if <code>pandera.dtypes.DataType</code> is an integer.
<code>pandera.dtypes.is_uint</code>	Return True if <code>pandera.dtypes.DataType</code> is an unsigned integer.
<code>pandera.dtypes.is_complex</code>	Return True if <code>pandera.dtypes.DataType</code> is a complex number.
<code>pandera.dtypes.is_numeric</code>	Return True if <code>pandera.dtypes.DataType</code> is a complex number.
<code>pandera.dtypes.is_bool</code>	Return True if <code>pandera.dtypes.DataType</code> is a boolean.
<code>pandera.dtypes.is_string</code>	Return True if <code>pandera.dtypes.DataType</code> is a string.
<code>pandera.dtypes.is_datetime</code>	Return True if <code>pandera.dtypes.DataType</code> is a datetime.
<code>pandera.dtypes.is_timedelta</code>	Return True if <code>pandera.dtypes.DataType</code> is a timedelta.
<code>pandera.dtypes.immutable</code>	<code>dataclasses.dataclass()</code> decorator with different default values: <code>frozen=True</code> , <code>init=False</code> , <code>repr=False</code> .

7.15.2.3.1 `pandera.dtypes.is_subdtype`

`pandera.dtypes.is_subdtype(arg1, arg2)`
Returns True if first argument is lower/equal in `DataType` hierarchy.

Return type `bool`

7.15.2.3.2 `pandera.dtypes.is_float`

`pandera.dtypes.is_float(pandera_dtype)`
Return True if `pandera.dtypes.DataType` is a float.
Return type `bool`

7.15.2.3.3 `pandera.dtypes.is_int`

`pandera.dtypes.is_int(pandera_dtype)`
Return True if `pandera.dtypes.DataType` is an integer.
Return type `bool`

7.15.2.3.4 `pandera.dtypes.is_uint`

`pandera.dtypes.is_uint(pandera_dtype)`
Return True if `pandera.dtypes.DataType` is an unsigned integer.
Return type `bool`

7.15.2.3.5 `pandera.dtypes.is_complex`

`pandera.dtypes.is_complex(pandera_dtype)`
Return True if `pandera.dtypes.DataType` is a complex number.
Return type `bool`

7.15.2.3.6 `pandera.dtypes.is_numeric`

`pandera.dtypes.is_numeric(pandera_dtype)`
Return True if `pandera.dtypes.DataType` is a complex number.
Return type `bool`

7.15.2.3.7 `pandera.dtypes.is_bool`

`pandera.dtypes.is_bool(pandera_dtype)`
Return True if `pandera.dtypes.DataType` is a boolean.
Return type `bool`

7.15.2.3.8 pandera.dtypes.is_string

`pandera.dtypes.is_string(pandera_dtype)`
Return True if `pandera.dtypes.DataType` is a string.
Return type `bool`

7.15.2.3.9 pandera.dtypes.is_datetime

`pandera.dtypes.is_datetime(pandera_dtype)`
Return True if `pandera.dtypes.DataType` is a datetime.
Return type `bool`

7.15.2.3.10 pandera.dtypes.is_timedelta

`pandera.dtypes.is_timedelta(pandera_dtype)`
Return True if `pandera.dtypes.DataType` is a timedelta.
Return type `bool`

7.15.2.3.11 pandera.dtypes.immutable

`pandera.dtypes.immutable(pandera_dtype_cls=None, **dataclass_kwargs)`
`dataclasses.dataclass()` decorator with different default values: `frozen=True`, `init=False`, `repr=False`.

In addition, `init=False` disables inherited `__init__` method to ensure the `DataType`'s default attributes are not altered during initialization.

Parameters

- **dtype** – `DataType` to decorate.
- **dataclass_kwargs** (Any) – Keywords arguments forwarded to `dataclasses.dataclass()`.

Return type `Union[Type[~_Dtype], Callable[[Type[~_Dtype]], Type[~_Dtype]]]`

Returns Immutable `DataType`

7.15.2.4 Engines

<code>pandera.engines.engine.Engine</code>	Base Engine metaclass.
<code>pandera.engines.numpy_engine.Engine</code>	Numpy data type engine.
<code>pandera.engines.pandas_engine.Engine</code>	Pandas data type engine.

7.15.2.4.1 pandera.engines.engine.Engine

class `pandera.engines.engine.Engine`(*name*, *bases*, *namespace*, ****kwargs**)

Base Engine metaclass.

Keep a registry of concrete Engines.

Methods

<code>dtype</code>	Convert input into a Pandera <code>DataType</code> object.
<code>get_registered_dtypes</code>	Return the <code>pandera.dtypes.DataTypes</code> registered with this engine.
<code>register_dtype</code>	Register a Pandera <code>DataType</code> with the engine, as class decorator.
<code>__call__</code>	Call self as a function.

7.15.2.4.1.1 pandera.engines.engine.Engine.dtype

`Engine.dtype`(*data_type*)

Convert input into a Pandera `DataType` object.

Return type `~_DataType`

7.15.2.4.1.2 pandera.engines.engine.Engine.get_registered_dtypes

`Engine.get_registered_dtypes`()

Return the `pandera.dtypes.DataTypes` registered with this engine.

Return type `List[Type[DataType]]`

7.15.2.4.1.3 pandera.engines.engine.Engine.register_dtype

`Engine.register_dtype`(*pandera_dtype_cls=None*, *****, *equivalents=None*)

Register a Pandera `DataType` with the engine, as class decorator.

Parameters

- **pandera_dtype** – The `DataType` to register.
- **equivalents** (`Optional[List[Any]]`) – Equivalent scalar data type classes or non-parametrized data type instances.

Note: The classmethod `from_parametrized_dtype` will also be registered. See [here](#) for more usage details.

Example

```

>>> import pandera as pa
>>>
>>> class MyDataType(pa.DataType):
...     pass
>>>
>>> class MyEngine(
...     metaclass=pa.engines.engine.Engine,
...     base_pandera_dtypes=MyDataType,
... ):
...     pass
>>>
>>> @MyEngine.register_dtype(equivalents=[bool])
... class MyBool(MyDataType):
...     pass

```

Return type `Callable`

7.15.2.4.1.4 `pandera.engines.engine.Engine.__call__`

`Engine.__call__(*args, **kwargs)`
Call self as a function.

7.15.2.4.2 `pandera.engines.numpy_engine.Engine`

class `pandera.engines.numpy_engine.Engine`
Numpy data type engine.

Methods

<code>dtype</code>	Convert input into a numpy-compatible Pandera <code>DataType</code> object.
--------------------	---

7.15.2.4.2.1 `pandera.engines.numpy_engine.Engine.dtype`

classmethod `Engine.dtype(data_type)`
Convert input into a numpy-compatible Pandera `DataType` object.

Return type `DataType`

7.15.2.4.3 `pandera.engines.pandas_engine.Engine`

class `pandera.engines.pandas_engine.Engine`

Pandas data type engine.

Methods

<code>dtype</code>	Convert input into a pandas-compatible Pandera <i>DataType</i> object.
<code>numpy_dtype</code>	Convert a Pandera dtype.

7.15.2.4.3.1 `pandera.engines.pandas_engine.Engine.dtype`

classmethod `Engine.dtype(data_type)`

Convert input into a pandas-compatible Pandera *DataType* object.

Return type `DataType`

7.15.2.4.3.2 `pandera.engines.pandas_engine.Engine.numpy_dtype`

classmethod `Engine.numpy_dtype(pandera_dtype)`

Convert a Pandera dtype.

Return type `dtype`

7.15.2.5 `PandasDtype Enum`

Warning: This class deprecated and will be removed from the pandera API in 0.9.0

<code>pandera.engines.pandas_engine.PandasDtype</code>	Enumerate all valid pandas data types.
--	--

7.15.2.5.1 `pandera.engines.pandas_engine.PandasDtype`

class `pandera.engines.pandas_engine.PandasDtype(value)`

Enumerate all valid pandas data types.

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.

Warning: This class is deprecated and will be removed in pandera v0.9.0. Use python types, pandas type string aliases, numpy dtypes, or pandas dtypes instead. See *Pandera Data Types* for details.

Examples

```

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

```

Attributes

Bool	"bool" numpy dtype
DateTime	"datetime64[ns]" numpy dtype
Timedelta	"timedelta64[ns]" numpy dtype
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
Object	"object" numpy dtype
Complex	"complex" numpy dtype
Complex64	"complex" numpy dtype
Complex128	"complex" numpy dtype
Complex256	"complex" numpy dtype
Category	pandas "categorical" datatype
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+

continues on next page

Table 98 – continued from previous page

UINT16	"UInt16" pandas dtype: pandas 0.24.0+
UINT32	"UInt32" pandas dtype: pandas 0.24.0+
UINT64	"UInt64" pandas dtype: pandas 0.24.0+
String	"str" numpy dtype
STRING	"string" pandas dtypes: pandas 1.0.0+.

7.15.3 Schema Models

7.15.3.1 Schema Model

`pandera.model.SchemaModel(*args, **kwargs)` Definition of a DataFrameSchema.

7.15.3.1.1 pandera.model.SchemaModel

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

Definition of a DataFrameSchema.

new in 0.5.0

See the *User Guide* for more.

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

Parameters

- **check_obj** (`pd.DataFrame`) – the dataframe to be validated.
- **head** – validate the first n rows. Rows overlapping with *tail* or *sample* are de-duplicated.
- **tail** – validate the last n rows. Rows overlapping with *head* or *sample* are de-duplicated.
- **sample** – validate a random sample of n rows. Rows overlapping with *head* or *tail* are de-duplicated.
- **random_state** – random seed for the *sample* argument.
- **lazy** – if True, lazily evaluates dataframe against all validation checks and raises a `SchemaErrors`. Otherwise, raise `SchemaError` as soon as one occurs.
- **inplace** – if True, applies coercion to the object of validation, otherwise creates a copy of the data.

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)

```

... })
>>>
>>> schema_withchecks = pa.DataFrameSchema({
...     "probability": pa.Column(
...         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(
...         str, [
...             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

```

Methods

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

7.15.3.1.1.1 pandera.model.SchemaModel.example

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

Create a hypothesis strategy for generating a DataFrame.

Parameters

- **size** (Optional[int]) – number of elements to generate
- **n_regex_columns** – number of regex columns to generate.

Return type DataFrameBase[~TSchemaModel]

Returns a strategy that generates pandas DataFrame objects.

7.15.3.1.1.2 `pandera.model.SchemaModel.strategy`

classmethod `SchemaModel.strategy(cls, *, size=None)`
 Create a hypothesis strategy for generating a DataFrame.

Parameters

- **size** (`Optional[int]`) – number of elements to generate
- **n_regex_columns** – number of regex columns to generate.

Return type `DataFrameBase[~TSchemaModel]`

Returns a strategy that generates pandas DataFrame objects.

7.15.3.1.1.3 `pandera.model.SchemaModel.to_schema`

classmethod `SchemaModel.to_schema()`
 Create `DataFrameSchema` from the `SchemaModel`.

Return type `DataFrameSchema`

7.15.3.1.1.4 `pandera.model.SchemaModel.to_yaml`

classmethod `SchemaModel.to_yaml(stream=None)`
 Convert `Schema` to yaml using `io.to_yaml`.

7.15.3.1.1.5 `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

- **check_obj** (`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 `DataFrameBase[~TSchemaModel]`

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(
...         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(
...         str, [
...             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

```

7.15.3.2 Model Components

<code>pandera.model_components.Field(*[, eq, ne, ...])</code>	Used to provide extra information about a field of a SchemaModel.
<code>pandera.model_components.check(*fields[, regex])</code>	Decorator to make SchemaModel method a column/index check function.
<code>pandera.model_components.dataframe_check([_fn])</code>	Decorator to make SchemaModel method a dataframe-wide check function.

7.15.3.2.1 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, unique=False,
                               allow_duplicates=None, 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 can contain null values.
- **unique** (`bool`) – Whether column values should be unique.
- **allow_duplicates** (`Optional[bool]`) – Whether or not column can contain duplicate values.

Warning: This option will be deprecated in 0.8.0. Use the `unique` argument instead.

- **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`

7.15.3.2.2 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]`

7.15.3.2.3 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]`

7.15.3.3 Typing

pandera.typing

Typing module.

7.15.3.3.1 pandera.typing

Typing module.

For backwards compatibility, pandas types are exposed to the top-level scope of the typing module.

Pandas object annotations

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

Dtype annotations

<code>Bool()</code>	Semantic representation of a boolean data type.
<code>DateTime</code>	alias of <code>pandera.dtypes.Timestamp</code>
<code>Timedelta()</code>	Semantic representation of a delta time data type.
<code>Category([categories, ordered])</code>	Semantic representation of a categorical data type.
<code>Float()</code>	Semantic representation of a floating data type.
<code>Float16()</code>	Semantic representation of a floating data type stored in 16 bits.
<code>Float32()</code>	Semantic representation of a floating data type stored in 32 bits.
<code>Float64()</code>	Semantic representation of a floating data type stored in 64 bits.
<code>Int()</code>	Semantic representation of an integer data type.
<code>Int8()</code>	Semantic representation of an integer data type stored in 8 bits.
<code>Int16()</code>	Semantic representation of an integer data type stored in 16 bits.
<code>Int32()</code>	Semantic representation of an integer data type stored in 32 bits.
<code>Int64()</code>	Semantic representation of an integer data type stored in 64 bits.
<code>UInt8()</code>	Semantic representation of an unsigned integer data type stored in 8 bits.
<code>UInt16()</code>	Semantic representation of an unsigned integer data type stored in 16 bits.
<code>UInt32()</code>	Semantic representation of an unsigned integer data type stored in 32 bits.
<code>UInt64()</code>	Semantic representation of an unsigned integer data type stored in 64 bits.
<code>INT8()</code>	Semantic representation of a <code>pandas.Int8Dtype</code> .
<code>INT16()</code>	Semantic representation of a <code>pandas.Int16Dtype</code> .
<code>INT32()</code>	Semantic representation of a <code>pandas.Int32Dtype</code> .
<code>INT64()</code>	Semantic representation of a <code>pandas.Int64Dtype</code> .
<code>UINT8()</code>	Semantic representation of a <code>pandas.UInt8Dtype</code> .
<code>UINT16()</code>	Semantic representation of a <code>pandas.UInt16Dtype</code> .
<code>UINT32()</code>	Semantic representation of a <code>pandas.UInt32Dtype</code> .
<code>UINT64()</code>	Semantic representation of a <code>pandas.UInt64Dtype</code> .
<code>Object()</code>	Semantic representation of a <code>numpy.object_</code> .
<code>String()</code>	Semantic representation of a string data type.
<code>STRING([storage])</code>	Semantic representation of a <code>pandas.StringDtype</code> .

7.15.3.4 Config

pandera.model.BaseConfig

Define DataFrameSchema-wide options.

7.15.3.4.1 pandera.model.BaseConfig

class `pandera.model.BaseConfig`

Bases: `object`

Define DataFrameSchema-wide options.

new in 0.5.0

Attributes

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

7.15.4 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.

7.15.4.1 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[int, str, 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(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.

7.15.4.2 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(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
>>>
```

(continues on next page)

(continued from previous page)

```

>>> 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.

7.15.4.3 `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

7.15.4.4 pandera.decorators.check_types

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

7.15.5 Schema Inference

<code>pandera.schema_inference.infer_schema</code>	Infer schema for pandas DataFrame or Series object.
--	---

7.15.5.1 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.

7.15.6 IO Utilities

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

<code>pandera.io.from_yaml</code>	Create <code>DataFrameSchema</code> from yaml file.
<code>pandera.io.to_yaml</code>	Write <code>DataFrameSchema</code> to yaml file.
<code>pandera.io.to_script</code>	Write <code>DataFrameSchema</code> to a python script.

7.15.6.1 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.

7.15.6.2 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.

7.15.6.3 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.

7.15.7 Data Synthesis Strategies

<code>pandera.strategies</code>	Generate synthetic data from a schema definition.
---------------------------------	---

7.15.7.1 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(pandera_dtype, strategy=None, *, checks=None, unique=False,
                                   name=None)
```

Create a data object describing a column in a DataFrame.

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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.
- **unique** (`bool`) – whether or not generated Series contains unique values.
- **name** (`Optional[str]`) – name of the Series.

Returns a `column` object.

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

Strategy to generate a pandas DataFrame.

Parameters

- **pandera_dtype** (`Optional[DataType]`) – `pandera.dtypes.DataType` 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.
- **unique** (`Optional[List[str]]`) – a list of column names that should be jointly unique.
- **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(pandera_dtype, strategy=None, *, value)
```

Strategy to generate a single value.

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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(pandera_dtype, strategy=None, *, checks=None)`

Strategy to generate elements of a column or index.

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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(pandera_dtype, strategy=None, *, min_value)`

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

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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(pandera_dtype, strategy=None, *, min_value)`

Strategy to generate values greater than a minimum value.

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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(pandera_dtype, strategy=None, *, min_value, max_value, include_min=True, include_max=True)`

Strategy to generate values within a particular range.

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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`(*pandera_dtype*, *strategy=None*, *, *checks=None*, *nullable=False*, *unique=False*, *name=None*, *size=None*)

Strategy to generate a pandas Index.

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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** (`bool`) – whether or not generated Series contains null values.
- **unique** (`bool`) – whether or not generated Series contains unique values.
- **name** (`Optional[str]`) – name of the Series.
- **size** (`Optional[int]`) – number of elements in the Series.

Returns hypothesis strategy.

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

Strategy to generate values within a finite set.

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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`(*pandera_dtype*, *strategy=None*, *, *max_value*)

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

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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`(*pandera_dtype*, *strategy=None*, *, *max_value*)
Strategy to generate values less than a maximum value.

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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`(*pandera_dtype=None*, *strategy=None*, *, *indexes=None*, *size=None*)

Strategy to generate a pandas MultiIndex object.

Parameters

- **pandera_dtype** (`Optional[DataType]`) – `pandera.dtypes.DataType` 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 `Index` objects.
- **size** (`Optional[int]`) – number of elements in the Series.

Returns hypothesis strategy.

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

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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`(*pandera_dtype*, *strategy=None*, *, *forbidden_values*)
Strategy to generate values excluding a set of forbidden values

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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** (`Union[dtype, DatetimeTZDtype]`) – 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(pandera_dtype, strategy=None, **kwargs)`
Strategy to generate data from a `pandera.dtypes.DataType`.

Parameters

- **pandera_dtype** (`DataType`) – `pandera.dtypes.DataType` 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(pandera_dtype, strategy=None, *, checks=None, nullable=False, unique=False, name=None, size=None)`

Strategy to generate a pandas Series.

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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** (`bool`) – whether or not generated Series contains null values.
- **unique** (`bool`) – whether or not generated Series contains unique values.
- **name** (`Optional[str]`) – name of the Series.

- **size** (*Optional[int]*) – number of elements in the Series.

Returns hypothesis strategy.

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

Strategy to generate strings that contain a particular pattern.

Parameters

- **pandera_dtype** (*Union[DataType, DataType]*) – `pandera.dtypes.DataType` 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(pandera_dtype, strategy=None, *, string)`

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

Parameters

- **pandera_dtype** (*Union[DataType, DataType]*) – `pandera.dtypes.DataType` 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(pandera_dtype, strategy=None, *, min_value, max_value)`

Strategy to generate strings of a particular length

Parameters

- **pandera_dtype** (*Union[DataType, DataType]*) – `pandera.dtypes.DataType` 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(pandera_dtype, strategy=None, *, pattern)`

Strategy to generate strings that patch a regex pattern.

Parameters

- **pandera_dtype** (*Union[DataType, DataType]*) – `pandera.dtypes.DataType` 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(pandera_dtype, strategy=None, *, string)`
Strategy to generate strings that start with a specific string pattern.

Parameters

- **pandera_dtype** (`Union[DataType, DataType]`) – `pandera.dtypes.DataType` 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.to_numpy_dtype(pandera_dtype)`
Convert a `DataType` to numpy dtype compatible with hypothesis.

`pandera.strategies.verify_dtype(pandera_dtype, schema_type, name)`
Verify that `pandera_dtype` argument is not `None`.

7.15.8 Extensions

`pandera.extensions`

pandera API extensions

7.15.8.1 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.

7.15.9 Errors

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

7.15.9.1 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.

7.15.9.2 `pandera.errors.SchemaErrors`

`class pandera.errors.SchemaErrors(schema_errors, data)`
Raised when multiple schema are lazily collected into one error.

7.15.9.3 `pandera.errors.SchemaInitError`

`class pandera.errors.SchemaInitError`
Raised when schema initialization fails.

7.15.9.4 `pandera.errors.SchemaDefinitionError`

`class pandera.errors.SchemaDefinitionError`
Raised when schema definition is invalid on object validation.

7.16 Contributing

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

7.16.1 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.

7.16.2 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.

First create your own fork of *pandera*, then clone it:

```
# replace <my-username> with your github username
git clone https://github.com/<my-username>/pandera.git
```

Once you’ve obtained a copy of the code, create a development environment that’s 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.

7.16.2.1 Environment Setup

7.16.2.1.1 Option 1: miniconda Setup

Install [miniconda](#), then run:

```
conda create -n pandera-dev python=3.8 # or any python version 3.7+
conda env update -n pandera-dev -f environment.yml
conda activate pandera-dev
pip install -e .
```

7.16.2.1.2 Option 2: virtualenv Setup

```
pip install virtualenv
virtualenv .venv/pandera-dev
source .venv/pandera-dev/bin/activate
pip install -r requirements-dev.txt
pip install -e .
```

7.16.2.1.3 Run Tests

```
pytest tests
```

7.16.2.1.4 Build Documentation Locally

```
make docs
```

7.16.2.1.5 Adding New Dependencies

To add new dependencies to the project, make sure to alter the *environment.yml* file. Then to sync the dependencies from the *environment.yml* file to the *requirements-dev.txt* run the following command

```
python scripts/generate_pip_deps_from_conda.py
```

Moreover to add new dependencies in *setup.py*, it is necessary to add it to the `_extras_require` dictionary.

7.16.2.1.6 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.

Make sure everything is working correctly by running

```
pre-commit run --all
```

7.16.2.2 Making Changes

Before making changes to the codebase or documentation, create a new branch with:

```
git checkout -b <my-branch>
```

We recommend following the branch-naming convention described in [Making Pull Requests](#).

7.16.2.3 Run the Full Test Suite Locally

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

```
# option 1: if you're working with conda (recommended)
$ make nox-conda

# option 2: if you're working with virtualenv
$ make nox
```

Option 2 assumes that you have python environments for all of the versions that pandera supports.

7.16.2.3.1 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
```

7.16.2.4 Project Releases

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

7.16.2.5 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`.

7.16.2.6 Contributing Bugfixes

Bugs are reported under the `bug` label, so if you find a bug create a new issue [here](#).

7.16.2.7 Contributing Enhancements

New feature issues can be found under the [enhancements](#) label. You can request a feature by creating a new issue [here](#).

7.16.2.8 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:

7.16.2.8.1 Bugfixes

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

7.16.2.8.2 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`

7.16.2.8.3 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!

7.16.2.9 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.

7.16.2.10 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=[
```

(continues on next page)

```
        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(
            ...,
        )
    },
    strict=True
)
```

HOW TO CITE

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

8.1 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 }  
}
```

8.2 Software Package

LICENSE AND CREDITS

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

INDICES AND TABLES

- `genindex`

PYTHON MODULE INDEX

p

`pandera.extensions`, 204
`pandera.strategies`, 198
`pandera.typing`, 190

Symbols

- `__call__` () (*pandera.checks.Check* method), 123
- `__call__` () (*pandera.dtypes.Bool* method), 134
- `__call__` () (*pandera.dtypes.Category* method), 138
- `__call__` () (*pandera.dtypes.Complex* method), 158
- `__call__` () (*pandera.dtypes.Complex128* method), 160
- `__call__` () (*pandera.dtypes.Complex256* method), 161
- `__call__` () (*pandera.dtypes.Complex64* method), 159
- `__call__` () (*pandera.dtypes.DataType* method), 133
- `__call__` () (*pandera.dtypes.Float* method), 139
- `__call__` () (*pandera.dtypes.Float128* method), 144
- `__call__` () (*pandera.dtypes.Float16* method), 140
- `__call__` () (*pandera.dtypes.Float32* method), 141
- `__call__` () (*pandera.dtypes.Float64* method), 143
- `__call__` () (*pandera.dtypes.Int* method), 145
- `__call__` () (*pandera.dtypes.Int16* method), 148
- `__call__` () (*pandera.dtypes.Int32* method), 149
- `__call__` () (*pandera.dtypes.Int64* method), 150
- `__call__` () (*pandera.dtypes.Int8* method), 146
- `__call__` () (*pandera.dtypes.String* method), 162
- `__call__` () (*pandera.dtypes.Timedelta* method), 136
- `__call__` () (*pandera.dtypes.Timestamp* method), 135
- `__call__` () (*pandera.dtypes.UInt* method), 151
- `__call__` () (*pandera.dtypes.UInt16* method), 154
- `__call__` () (*pandera.dtypes.UInt32* method), 155
- `__call__` () (*pandera.dtypes.UInt64* method), 156
- `__call__` () (*pandera.dtypes.UInt8* method), 153
- `__call__` () (*pandera.engines.engine.Engine* method), 182
- `__call__` () (*pandera.engines.numpy_engine.Object* method), 178
- `__call__` () (*pandera.engines.pandas_engine.BOOL* method), 164
- `__call__` () (*pandera.engines.pandas_engine.INT16* method), 167
- `__call__` () (*pandera.engines.pandas_engine.INT32* method), 168
- `__call__` () (*pandera.engines.pandas_engine.INT64* method), 169
- `__call__` () (*pandera.engines.pandas_engine.INT8* method), 165
- `__call__` () (*pandera.engines.pandas_engine.STRING* method), 176
- `__call__` () (*pandera.engines.pandas_engine.UINT16* method), 172
- `__call__` () (*pandera.engines.pandas_engine.UINT32* method), 173
- `__call__` () (*pandera.engines.pandas_engine.UINT64* method), 175
- `__call__` () (*pandera.engines.pandas_engine.UINT8* method), 171
- `__call__` () (*pandera.hypotheses.Hypothesis* method), 130
- `__call__` () (*pandera.schema_components.Column* method), 107
- `__call__` () (*pandera.schema_components.Index* method), 110
- `__call__` () (*pandera.schema_components.MultiIndex* method), 114
- `__call__` () (*pandera.schemas.DataFrameSchema* method), 99
- `__call__` () (*pandera.schemas.SeriesSchema* method), 102
- `__init__` () (*pandera.dtypes.Bool* method), 134
- `__init__` () (*pandera.dtypes.Category* method), 137
- `__init__` () (*pandera.dtypes.Complex* method), 157
- `__init__` () (*pandera.dtypes.Complex128* method), 160
- `__init__` () (*pandera.dtypes.Complex256* method), 161
- `__init__` () (*pandera.dtypes.Complex64* method), 158
- `__init__` () (*pandera.dtypes.DataType* method), 132
- `__init__` () (*pandera.dtypes.Float* method), 138
- `__init__` () (*pandera.dtypes.Float128* method), 143
- `__init__` () (*pandera.dtypes.Float16* method), 140
- `__init__` () (*pandera.dtypes.Float32* method), 141
- `__init__` () (*pandera.dtypes.Float64* method), 142
- `__init__` () (*pandera.dtypes.Int* method), 145
- `__init__` () (*pandera.dtypes.Int16* method), 147
- `__init__` () (*pandera.dtypes.Int32* method), 148
- `__init__` () (*pandera.dtypes.Int64* method), 150
- `__init__` () (*pandera.dtypes.Int8* method), 146
- `__init__` () (*pandera.dtypes.String* method), 162
- `__init__` () (*pandera.dtypes.Timedelta* method), 136
- `__init__` () (*pandera.dtypes.Timestamp* method), 135
- `__init__` () (*pandera.dtypes.UInt* method), 151

- `__init__()` (*pandera.dtypes.UInt16 method*), 153
 - `__init__()` (*pandera.dtypes.UInt32 method*), 155
 - `__init__()` (*pandera.dtypes.UInt64 method*), 156
 - `__init__()` (*pandera.dtypes.UInt8 method*), 152
 - `__init__()` (*pandera.engines.numpy_engine.Object method*), 177
 - `__init__()` (*pandera.engines.pandas_engine.BOOL method*), 164
 - `__init__()` (*pandera.engines.pandas_engine.INT16 method*), 166
 - `__init__()` (*pandera.engines.pandas_engine.INT32 method*), 168
 - `__init__()` (*pandera.engines.pandas_engine.INT64 method*), 169
 - `__init__()` (*pandera.engines.pandas_engine.INT8 method*), 165
 - `__init__()` (*pandera.engines.pandas_engine.STRING method*), 176
 - `__init__()` (*pandera.engines.pandas_engine.UINT16 method*), 172
 - `__init__()` (*pandera.engines.pandas_engine.UINT32 method*), 173
 - `__init__()` (*pandera.engines.pandas_engine.UINT64 method*), 174
 - `__init__()` (*pandera.engines.pandas_engine.UINT8 method*), 170
 - `__init__()` (*pandera.hypotheses.Hypothesis method*), 126
 - `__init__()` (*pandera.schema_components.Column method*), 104
 - `__init__()` (*pandera.schema_components.MultiIndex method*), 111
 - `__init__()` (*pandera.schemas.DataFrameSchema method*), 87
 - `__init__()` (*pandera.schemas.SeriesSchema method*), 101
- ## A
- `add_columns()` (*pandera.schemas.DataFrameSchema method*), 88
- ## B
- `BaseConfig` (*class in pandera.model*), 192
 - `Bool` (*class in pandera.dtypes*), 133
 - `BOOL` (*class in pandera.engines.pandas_engine*), 163
- ## C
- `Category` (*class in pandera.dtypes*), 137
 - `Check` (*class in pandera.checks*), 114
 - `check()` (*in module pandera.model_components*), 190
 - `check()` (*pandera.dtypes.Bool method*), 134
 - `check()` (*pandera.dtypes.Category method*), 137
 - `check()` (*pandera.dtypes.Complex method*), 157
 - `check()` (*pandera.dtypes.Complex128 method*), 160
 - `check()` (*pandera.dtypes.Complex256 method*), 161
 - `check()` (*pandera.dtypes.Complex64 method*), 158
 - `check()` (*pandera.dtypes.DataType method*), 132
 - `check()` (*pandera.dtypes.Float method*), 139
 - `check()` (*pandera.dtypes.Float128 method*), 143
 - `check()` (*pandera.dtypes.Float16 method*), 140
 - `check()` (*pandera.dtypes.Float32 method*), 141
 - `check()` (*pandera.dtypes.Float64 method*), 142
 - `check()` (*pandera.dtypes.Int method*), 145
 - `check()` (*pandera.dtypes.Int16 method*), 147
 - `check()` (*pandera.dtypes.Int32 method*), 148
 - `check()` (*pandera.dtypes.Int64 method*), 150
 - `check()` (*pandera.dtypes.Int8 method*), 146
 - `check()` (*pandera.dtypes.String method*), 162
 - `check()` (*pandera.dtypes.Timedelta method*), 136
 - `check()` (*pandera.dtypes.Timestamp method*), 135
 - `check()` (*pandera.dtypes.UInt method*), 151
 - `check()` (*pandera.dtypes.UInt16 method*), 153
 - `check()` (*pandera.dtypes.UInt32 method*), 155
 - `check()` (*pandera.dtypes.UInt64 method*), 156
 - `check()` (*pandera.dtypes.UInt8 method*), 152
 - `check()` (*pandera.engines.numpy_engine.Object method*), 177
 - `check()` (*pandera.engines.pandas_engine.BOOL method*), 164
 - `check()` (*pandera.engines.pandas_engine.INT16 method*), 166
 - `check()` (*pandera.engines.pandas_engine.INT32 method*), 168
 - `check()` (*pandera.engines.pandas_engine.INT64 method*), 169
 - `check()` (*pandera.engines.pandas_engine.INT8 method*), 165
 - `check()` (*pandera.engines.pandas_engine.STRING method*), 176
 - `check()` (*pandera.engines.pandas_engine.UINT16 method*), 172
 - `check()` (*pandera.engines.pandas_engine.UINT32 method*), 173
 - `check()` (*pandera.engines.pandas_engine.UINT64 method*), 174
 - `check()` (*pandera.engines.pandas_engine.UINT8 method*), 170
 - `check_input()` (*in module pandera.decorators*), 192
 - `check_io()` (*in module pandera.decorators*), 195
 - `check_output()` (*in module pandera.decorators*), 194
 - `check_types()` (*in module pandera.decorators*), 196
 - `checks` (*pandera.io.FrictionlessFieldParser property*), 70
 - `CheckType` (*class in pandera.extensions*), 204
 - `coerce` (*pandera.io.FrictionlessFieldParser property*), 70
 - `coerce()` (*pandera.dtypes.Bool method*), 134
 - `coerce()` (*pandera.dtypes.Category method*), 137

- `coerce()` (*pandera.dtypes.Complex method*), 157
`coerce()` (*pandera.dtypes.Complex128 method*), 160
`coerce()` (*pandera.dtypes.Complex256 method*), 161
`coerce()` (*pandera.dtypes.Complex64 method*), 159
`coerce()` (*pandera.dtypes.DataType method*), 133
`coerce()` (*pandera.dtypes.Float method*), 139
`coerce()` (*pandera.dtypes.Float128 method*), 144
`coerce()` (*pandera.dtypes.Float16 method*), 140
`coerce()` (*pandera.dtypes.Float32 method*), 141
`coerce()` (*pandera.dtypes.Float64 method*), 142
`coerce()` (*pandera.dtypes.Int method*), 145
`coerce()` (*pandera.dtypes.Int16 method*), 147
`coerce()` (*pandera.dtypes.Int32 method*), 149
`coerce()` (*pandera.dtypes.Int64 method*), 150
`coerce()` (*pandera.dtypes.Int8 method*), 146
`coerce()` (*pandera.dtypes.String method*), 162
`coerce()` (*pandera.dtypes.Timedelta method*), 136
`coerce()` (*pandera.dtypes.Timestamp method*), 135
`coerce()` (*pandera.dtypes.UInt method*), 151
`coerce()` (*pandera.dtypes.UInt16 method*), 154
`coerce()` (*pandera.dtypes.UInt32 method*), 155
`coerce()` (*pandera.dtypes.UInt64 method*), 156
`coerce()` (*pandera.dtypes.UInt8 method*), 152
`coerce()` (*pandera.engines.numpy_engine.Object method*), 177
`coerce()` (*pandera.engines.pandas_engine.BOOL method*), 164
`coerce()` (*pandera.engines.pandas_engine.INT16 method*), 166
`coerce()` (*pandera.engines.pandas_engine.INT32 method*), 168
`coerce()` (*pandera.engines.pandas_engine.INT64 method*), 169
`coerce()` (*pandera.engines.pandas_engine.INT8 method*), 165
`coerce()` (*pandera.engines.pandas_engine.STRING method*), 176
`coerce()` (*pandera.engines.pandas_engine.UINT16 method*), 172
`coerce()` (*pandera.engines.pandas_engine.UINT32 method*), 173
`coerce()` (*pandera.engines.pandas_engine.UINT64 method*), 174
`coerce()` (*pandera.engines.pandas_engine.UINT8 method*), 170
`coerce_dtype()` (*pandera.schema_components.Column method*), 106
`coerce_dtype()` (*pandera.schema_components.MultiIndex method*), 112
`coerce_dtype()` (*pandera.schemas.DataFrameSchema method*), 89
`Column` (*class in pandera.schema_components*), 103
`column_strategy()` (*in module pandera.strategies*), 198
`Complex` (*class in pandera.dtypes*), 156
`Complex128` (*class in pandera.dtypes*), 159
`Complex256` (*class in pandera.dtypes*), 160
`Complex64` (*class in pandera.dtypes*), 158
D
`dataframe_check()` (*in module pandera.model_components*), 190
`dataframe_strategy()` (*in module pandera.strategies*), 198
`DataFrameSchema` (*class in pandera.schemas*), 84
`DataType` (*class in pandera.dtypes*), 132
`DateTime` (*in module pandera.dtypes*), 135
`dtype` (*pandera.io.FrictionlessFieldParser property*), 71
`dtype()` (*pandera.engines.engine.Engine method*), 181
`dtype()` (*pandera.engines.numpy_engine.Engine class method*), 182
`dtype()` (*pandera.engines.pandas_engine.Engine class method*), 183
E
`ELEMENT_WISE` (*pandera.extensions.CheckType attribute*), 204
`Engine` (*class in pandera.engines.engine*), 181
`Engine` (*class in pandera.engines.numpy_engine*), 182
`Engine` (*class in pandera.engines.pandas_engine*), 183
`eq()` (*pandera.checks.Check class method*), 117
`eq_strategy()` (*in module pandera.strategies*), 198
`equal_to()` (*pandera.checks.Check class method*), 117
`example()` (*pandera.model.SchemaModel class method*), 186
`example()` (*pandera.schema_components.Column method*), 106
`example()` (*pandera.schema_components.Index method*), 109
`example()` (*pandera.schema_components.MultiIndex method*), 113
`example()` (*pandera.schemas.DataFrameSchema method*), 89
F
`Field()` (*in module pandera.model_components*), 189
`field_element_strategy()` (*in module pandera.strategies*), 199
`Float` (*class in pandera.dtypes*), 138
`Float128` (*class in pandera.dtypes*), 143
`Float16` (*class in pandera.dtypes*), 139
`Float32` (*class in pandera.dtypes*), 140
`Float64` (*class in pandera.dtypes*), 141
`FrictionlessFieldParser` (*class in pandera.io*), 70
`from_frictionless_schema()` (*in module pandera.io*), 69

- from_parametrized_dtype() (*pandera.engines.pandas_engine.STRING* class method), 176
 from_yaml() (*in module pandera.io*), 197
 from_yaml() (*pandera.schemas.DataFrameSchema* class method), 89
- ## G
- ge() (*pandera.checks.Check* class method), 118
 ge_strategy() (*in module pandera.strategies*), 199
 get_dtypes() (*pandera.schemas.DataFrameSchema* method), 90
 get_regex_columns() (*pandera.schema_components.Column* method), 106
 get_registered_dtypes() (*pandera.engines.engine.Engine* method), 181
 greater_than() (*pandera.checks.Check* class method), 118
 greater_than_or_equal_to() (*pandera.checks.Check* class method), 118
 GROUPBY (*pandera.extensions.CheckType* attribute), 204
 gt() (*pandera.checks.Check* class method), 119
 gt_strategy() (*in module pandera.strategies*), 199
- ## H
- Hypothesis (*class in pandera.hypotheses*), 124
- ## I
- immutable() (*in module pandera.dtypes*), 180
 in_range() (*pandera.checks.Check* class method), 119
 in_range_strategy() (*in module pandera.strategies*), 199
 Index (*class in pandera.schema_components*), 107
 index_strategy() (*in module pandera.strategies*), 200
 infer_schema() (*in module pandera.schema_inference*), 196
 Int (*class in pandera.dtypes*), 144
 Int16 (*class in pandera.dtypes*), 146
 INT16 (*class in pandera.engines.pandas_engine*), 166
 Int32 (*class in pandera.dtypes*), 148
 INT32 (*class in pandera.engines.pandas_engine*), 167
 Int64 (*class in pandera.dtypes*), 149
 INT64 (*class in pandera.engines.pandas_engine*), 168
 Int8 (*class in pandera.dtypes*), 145
 INT8 (*class in pandera.engines.pandas_engine*), 164
 is_bool() (*in module pandera.dtypes*), 179
 is_complex() (*in module pandera.dtypes*), 179
 is_datetime() (*in module pandera.dtypes*), 180
 is_float() (*in module pandera.dtypes*), 179
 is_int() (*in module pandera.dtypes*), 179
 is_numeric() (*in module pandera.dtypes*), 179
 is_string() (*in module pandera.dtypes*), 180
 is_subdtype() (*in module pandera.dtypes*), 178
 is_timedelta() (*in module pandera.dtypes*), 180
 is_uint() (*in module pandera.dtypes*), 179
 isin() (*pandera.checks.Check* class method), 119
 isin_strategy() (*in module pandera.strategies*), 200
- ## L
- le() (*pandera.checks.Check* class method), 120
 le_strategy() (*in module pandera.strategies*), 200
 less_than() (*pandera.checks.Check* class method), 120
 less_than_or_equal_to() (*pandera.checks.Check* class method), 120
 lt() (*pandera.checks.Check* class method), 121
 lt_strategy() (*in module pandera.strategies*), 200
- ## M
- module
 pandera.extensions, 204
 pandera.strategies, 198
 pandera.typing, 190
 MultiIndex (*class in pandera.schema_components*), 110
 multiindex_strategy() (*in module pandera.strategies*), 201
- ## N
- ne() (*pandera.checks.Check* class method), 121
 ne_strategy() (*in module pandera.strategies*), 201
 not_equal_to() (*pandera.checks.Check* class method), 121
 notin() (*pandera.checks.Check* class method), 122
 notin_strategy() (*in module pandera.strategies*), 201
 nullable (*pandera.io.FrictionlessFieldParser* property), 71
 numpy_complex_dtypes() (*in module pandera.strategies*), 201
 numpy_dtype() (*pandera.engines.pandas_engine.Engine* class method), 183
 numpy_time_dtypes() (*in module pandera.strategies*), 202
- ## O
- Object (*class in pandera.engines.numpy_engine*), 176
 one_sample_ttest() (*pandera.hypotheses.Hypothesis* class method), 128
- ## P
- pandas_dtype_strategy() (*in module pandera.strategies*), 202
 PandasDtype (*class in pandera.engines.pandas_engine*), 183
pandera.extensions
 module, 204
pandera.strategies
 module, 198

pandera.typing
module, 190

R

regex (*pandera.io.FrictionlessFieldParser* property), 71
 register_check_method() (in module *pandera.extensions*), 204
 register_check_strategy() (in module *pandera.strategies*), 202
 register_dtype() (*pandera.engines.engine.Engine* method), 181
 remove_columns() (*pandera.schemas.DataFrameSchema* method), 90
 rename_columns() (*pandera.schemas.DataFrameSchema* method), 91
 required (*pandera.io.FrictionlessFieldParser* property), 71
 reset_index() (*pandera.schemas.DataFrameSchema* method), 92

S

SchemaDefinitionError (class in *pandera.errors*), 206
 SchemaError (class in *pandera.errors*), 205
 SchemaErrors (class in *pandera.errors*), 206
 SchemaInitError (class in *pandera.errors*), 206
 SchemaModel (class in *pandera.model*), 185
 select_columns() (*pandera.schemas.DataFrameSchema* method), 93
 series_strategy() (in module *pandera.strategies*), 202
 SeriesSchema (class in *pandera.schemas*), 99
 set_index() (*pandera.schemas.DataFrameSchema* method), 94
 set_name() (*pandera.schema_components.Column* method), 106
 str_contains() (*pandera.checks.Check* class method), 122
 str_contains_strategy() (in module *pandera.strategies*), 203
 str_endswith() (*pandera.checks.Check* class method), 122
 str_endswith_strategy() (in module *pandera.strategies*), 203
 str_length() (*pandera.checks.Check* class method), 123
 str_length_strategy() (in module *pandera.strategies*), 203
 str_matches() (*pandera.checks.Check* class method), 123
 str_matches_strategy() (in module *pandera.strategies*), 203

str_startswith() (*pandera.checks.Check* class method), 123
 str_startswith_strategy() (in module *pandera.strategies*), 203
 strategy() (*pandera.model.SchemaModel* class method), 187
 strategy() (*pandera.schema_components.Column* method), 106
 strategy() (*pandera.schema_components.Index* method), 109
 strategy() (*pandera.schema_components.MultiIndex* method), 113
 strategy() (*pandera.schemas.DataFrameSchema* method), 95
 strategy_component() (*pandera.schema_components.Column* method), 106
 strategy_component() (*pandera.schema_components.Index* method), 109
 String (class in *pandera.dtypes*), 161
 STRING (class in *pandera.engines.pandas_engine*), 175

T

Timedelta (class in *pandera.dtypes*), 135
 Timestamp (class in *pandera.dtypes*), 134
 to_numpy_dtype() (in module *pandera.strategies*), 204
 to_pandera_column() (*pandera.io.FrictionlessFieldParser* method), 71
 to_schema() (*pandera.model.SchemaModel* class method), 187
 to_script() (in module *pandera.io*), 197
 to_script() (*pandera.schemas.DataFrameSchema* method), 96
 to_yaml() (in module *pandera.io*), 197
 to_yaml() (*pandera.model.SchemaModel* class method), 187
 to_yaml() (*pandera.schemas.DataFrameSchema* method), 96
 try_coerce() (*pandera.dtypes.Bool* method), 134
 try_coerce() (*pandera.dtypes.Category* method), 138
 try_coerce() (*pandera.dtypes.Complex* method), 157
 try_coerce() (*pandera.dtypes.Complex128* method), 160
 try_coerce() (*pandera.dtypes.Complex256* method), 161
 try_coerce() (*pandera.dtypes.Complex64* method), 159
 try_coerce() (*pandera.dtypes.DataType* method), 133
 try_coerce() (*pandera.dtypes.Float* method), 139
 try_coerce() (*pandera.dtypes.Float128* method), 144
 try_coerce() (*pandera.dtypes.Float16* method), 140
 try_coerce() (*pandera.dtypes.Float32* method), 141

`try_coerce()` (*pandera.dtypes.Float64* method), 142
`try_coerce()` (*pandera.dtypes.Int* method), 145
`try_coerce()` (*pandera.dtypes.Int16* method), 147
`try_coerce()` (*pandera.dtypes.Int32* method), 149
`try_coerce()` (*pandera.dtypes.Int64* method), 150
`try_coerce()` (*pandera.dtypes.Int8* method), 146
`try_coerce()` (*pandera.dtypes.String* method), 162
`try_coerce()` (*pandera.dtypes.Timedelta* method), 136
`try_coerce()` (*pandera.dtypes.Timestamp* method), 135
`try_coerce()` (*pandera.dtypes.UInt* method), 151
`try_coerce()` (*pandera.dtypes.UInt16* method), 154
`try_coerce()` (*pandera.dtypes.UInt32* method), 155
`try_coerce()` (*pandera.dtypes.UInt64* method), 156
`try_coerce()` (*pandera.dtypes.UInt8* method), 152
`try_coerce()` (*pandera.engines.numpy_engine.Object* method), 177
`try_coerce()` (*pandera.engines.pandas_engine.BOOL* method), 164
`try_coerce()` (*pandera.engines.pandas_engine.INT16* method), 167
`try_coerce()` (*pandera.engines.pandas_engine.INT32* method), 168
`try_coerce()` (*pandera.engines.pandas_engine.INT64* method), 169
`try_coerce()` (*pandera.engines.pandas_engine.INT8* method), 165
`try_coerce()` (*pandera.engines.pandas_engine.STRING* method), 176
`try_coerce()` (*pandera.engines.pandas_engine.UINT16* method), 172
`try_coerce()` (*pandera.engines.pandas_engine.UINT32* method), 173
`try_coerce()` (*pandera.engines.pandas_engine.UINT64* method), 175
`try_coerce()` (*pandera.engines.pandas_engine.UINT8* method), 171
`two_sample_ttest()` (*pandera.hypotheses.Hypothesis* class method), 129

U

`UInt` (class in *pandera.dtypes*), 150
`UInt16` (class in *pandera.dtypes*), 153
`UINT16` (class in *pandera.engines.pandas_engine*), 171
`UInt32` (class in *pandera.dtypes*), 154
`UINT32` (class in *pandera.engines.pandas_engine*), 172
`UInt64` (class in *pandera.dtypes*), 155
`UINT64` (class in *pandera.engines.pandas_engine*), 174
`UInt8` (class in *pandera.dtypes*), 151
`UINT8` (class in *pandera.engines.pandas_engine*), 170
`unique` (*pandera.io.FrictionlessFieldParser* property), 71
`update_column()` (*pandera.schemas.DataFrameSchema* method), 96

`update_columns()` (*pandera.schemas.DataFrameSchema* method), 97

V

`validate()` (*pandera.model.SchemaModel* class method), 187
`validate()` (*pandera.schema_components.Column* method), 107
`validate()` (*pandera.schema_components.Index* method), 109
`validate()` (*pandera.schema_components.MultiIndex* method), 113
`validate()` (*pandera.schemas.DataFrameSchema* method), 98
`validate()` (*pandera.schemas.SeriesSchema* method), 101
`VECTORIZED` (*pandera.extensions.CheckType* attribute), 204
`verify_dtype()` (in module *pandera.strategies*), 204