schemable Documentation

Release 0.5.0

Derrick Gilland

Aug 17, 2018

Contents

Schemable is a schema parsing and validation library that let's you define schemas simply using dictionaries, lists, types, and callables.

CHAPTER 1

Links

- Project: https://github.com/dgilland/schemable
- Documentation: https://schemable.readthedocs.io
- PyPI: https://pypi.python.org/pypi/schemable/
- TravisCI: https://travis-ci.org/dgilland/schemable

CHAPTER 2

Features

- Simple schema definitions using `dict`, `list`, and `type` objects
- Complex schema definitions using `Any`, `All`, `As`, and predicates
- Detailed validation error messages
- Partial data loading on validation failure
- Strict and non-strict parsing modes
- Python 3.4+

CHAPTER 3

Quickstart

Install using pip:

```
pip install schemable
```

Define a schema using `dict` and `list` objects:

```
from schemable import Schema, All, Any, As, Optional, SchemaError

user_schema = Schema({
    'name': str,
    'email': All(str, lambda email: len(email) > 3 and '@' in email),
    'active': bool,
    'settings': {
        Optional('theme'): str,
        Optional('language', default='en'): str,
        Optional('volume'): int,
        str: str
    },
    'aliases': [str],
    'phone': All(str,
                 As(lambda phone: ''.join(filter(str.isdigit, phone))),
                 lambda phone: 10 <= len(phone) <= 15),
    'addresses': [{
        'street_addr1': str,
        Optional('street_addr2', default=None): Any(str, None),
        'city': str,
        'state': str,
        'country': str,
        'zip_code': str
    }]
})
```

Then validate and load by passing data to `user_schema()`:

```
# Fail!
result = user_schema({
    'name': 'Bob Smith',
    'email': 'bob.example.com',
    'active': 1,
    'settings': {
        'theme': False,
        'extra_setting1': 'val1',
        'extra_setting2': True
    },
    'phone': 1234567890,
    'addresses': [
        {'street_addr1': '123 Lane',
         'city': 'City',
         'state': 'ST',
         'country': 'US',
         'zip_code': 11000}
    ]
})

print(result)
# SchemaResult(
#     data={'name': 'Bob Smith',
#           'settings': {'extra_setting1': 'val1',
#                        'language': 'en'}
#           'addresses': [{'street_addr1': '123 Lane',
#                          'city': 'City',
#                          'state': 'ST',
#                          'country': 'US',
#                          'street_addr2': None}]},
#     errors={'email': "bad value: <lambda>('bob.example.com') should evaluate to True
↪",
#             'active': 'bad value: type error, expected bool but found int',
#             'settings': {'theme': 'bad value: type error, expected str but found␣
↪bool',
#                          'extra_setting2': 'bad value: type error, expected str but␣
↪found bool'},
#             'phone': 'bad value: type error, expected str but found int',
#             'addresses': {0: {'zip_code': 'bad value: type error, expected str but␣
↪found int'}},
#             'aliases': 'missing required key'})

# Fail!
result = user_schema({
    'name': 'Bob Smith',
    'email': 'bob@example.com',
    'active': True,
    'settings': {
        'theme': False,
        'extra_setting1': 'val1',
        'extra_setting2': 'val2'
    },
    'phone': '123-456-789',
    'addresses': [
        {'street_addr1': '123 Lane',
         'city': 'City',
         'state': 'ST',
```

(continues on next page)

(continued from previous page)

```
        'country': 'US',
        'zip_code': '11000'}
    ]
})

print(result)
# SchemaResult(
#     data={'name': 'Bob Smith',
#           'email': 'bob@example.com',
#           'active': True,
#           'settings': {'extra_setting1': 'val1',
#                        'extra_setting2': 'val2',
#                        'language': 'en'},
#           'addresses': [{'street_addr1': '123 Lane',
#                          'city': 'City',
#                          'state': 'ST',
#                          'country': 'US',
#                          'zip_code': '11000',
#                          'street_addr2': None}]},
#     errors={'settings': {'theme': 'bad value: type error, expected str but found␣
↪bool'},
#             'phone': "bad value: <lambda>('123456789') should evaluate to True",
#             'aliases': 'missing required key'})
```

Or can raise an exception on validation failure instead of returning results:

```
# Fail strictly!
try:
    user_schema({
        'name': 'Bob Smith',
        'email': 'bob@example.com',
        'active': True,
        'settings': {
            'theme': False,
            'extra_setting1': 'val1',
            'extra_setting2': 'val2'
        },
        'phone': '123-456-789',
        'addresses': [
            {'street_addr1': '123 Lane',
             'city': 'City',
             'state': 'ST',
             'country': 'US',
             'zip_code': '11000'}
        ]
    }, strict=True)
except SchemaError as exc:
    print(exc)
    # Schema validation failed: \
    # {'settings': {'theme': 'bad value: type error, expected str but found bool'}, \
    # 'phone': "bad value: <lambda>('123456789') should evaluate to True", \
    # 'aliases': 'missing required key'}
```

Access the parsed data after successful validation:

```
# Pass!
result = user_schema({
```

(continues on next page)

(continued from previous page)

```
    'name': 'Bob Smith',
    'email': 'bob@example.com',
    'active': True,
    'settings': {
        'theme': 'dark',
        'extra_setting1': 'val1',
        'extra_setting2': 'val2'
    },
    'phone': '123-456-7890',
    'aliases': [],
    'addresses': [
        {'street_addr1': '123 Lane',
         'city': 'City',
         'state': 'ST',
         'country': 'US',
         'zip_code': '11000'}
    ]
})

print(result)
# SchemaResult(
#     data={'name': 'Bob Smith',
#           'email': 'bob@example.com',
#           'active': True,
#           'settings': {'theme': 'dark',
#                        'extra_setting1': 'val1',
#                        'extra_setting2': 'val2',
#                        'language': 'en'},
#           'phone': '1234567890',
#           'aliases': [],
#           'addresses': [{'street_addr1': '123 Lane',
#                          'city': 'City',
#                          'state': 'ST',
#                          'country': 'US',
#                          'zip_code': '11000',
#                          'street_addr2': None}]},
#     errors={})
```

For more details, please see the full documentation at https://schemable.readthedocs.io.

CHAPTER 4

Guide

4.1 Installation

schemable requires Python >= 3.4.

To install from PyPI:

```
pip install schemable
```

4.2 User Guide

Schemas are defined using the *Schema* class which returns a callable object that can then be used to validate and load data:

```
from schemable import Schema, SchemaResult

schema = Schema([str])
result = schema(['a', 'b', 'c'])

assert isinstance(result, SchemaResult)
assert hasattr(result, 'data')
assert hasattr(result, 'error')
```

The return from a schema call is a *SchemaResult* instance that contains two attributes: `data` and `errors`. The `data` object defaults to `None` when nothing could be successfully validated. It may also contain partially loaded data when some validation passed but other validation failed:

```
from schemable import Schema

schema = Schema({str: {str: {str: int}}})
schema({'a': {'b': {'c': 1}},
        'aa': {'bb': {'cc': 'dd'}}})
```

(continues on next page)

(continued from previous page)

```
# SchemaResult(
#     data={'a': {'b': {'c': 1}}},
#     errors={'aa': {'bb': {'cc': 'bad value: type error, expected int but found str'}
↪}})
```

The `errors` attribute will either be a dictionary mapping of errors (when the top-level schema is a `dict` or `list`) with keys corresponding to each point of failure or a string error message (when the top-level schema is *not* a `dict` or `list`). If there are no errors, then *`SchemaResult.errors`* will be either `{}` or `None`. The `errors` dictionary can span multiple "levels" and `list` indexes are treated as integer keys:

```
from schemable import Schema

schema = Schema({str: [int]})
schema({'a': [1, 2, '3', 4, '5'],
        'b': True})
# SchemaResult(
#     data={'a': [1, 2, 4]},
#     errors={'a': {2: 'bad value: type error, expected int but found str',
#                   4: 'bad value: type error, expected int but found str'},
#             'b': 'bad value: type error, expected list but found bool'})
```

By default, schemas are evaulated in non-strict mode which always returns a *`SchemaResult`* instance whether validation passed or failed. However, in strict mode the exception *`SchemaError`* will be raised instead.

There are two ways to set strict mode:

1. Set `strict=True` when creating a *`Schema`* object (i.e., `Schema(..., strict=True)`)
2. Set `strict=True` when evaulating a schema (i.e. `schema(..., strict=True)`)

TIP: If *`Schema`* was created with `strict=True`, use `schema(..., strict=False)` to evaulate the schema in non-strict mode.

```
from schemable import Schema

# Default to strict mode when evaulated.
schema = Schema({str: [int]}, strict=True)
schema({'a': [1, 2, '3', 4, '5'],
        'b': True})
# Traceback (most recent call last):
# ...
# SchemaError: Schema validation failed: {'a': {2: 'bad value: type error, expected␣
↪int but found str', 4: 'bad value: type error, expected int but found str'}, 'b':
↪'bad value: type error, expected list but found bool'}

# disable with schema(..., strict=False)

# Or use strict on a per-evaulation basis
schema = Schema({str: [int]})
schema({'a': [1, 2, '3', 4, '5'],
        'b': True},
       strict=True)
```

4.2.1 Validation

Schemable is able to validate against the following:

- types (using `type` objects like `str`, `int`, `bool`, etc.)
- raw values (like `5`, `'foo'`, etc.)
- dicts (using `dict` objects)
- lists (using `list` objects; applies schema object to all list items)
- nested schemas (using `dict`, `list`, or *`Schema`*)
- predicates (using callables that return a boolean value or raise an exception)
- all predicates (using *`All`*)
- any predicate (using *`Any`*)

Values

Validate against values:

```
from schemable import Schema

schema = Schema(5)
schema(5)
# SchemaResult(data=5, errors=None)

schema = Schema({'a': 5})
schema({'a': 5})
# SchemaResult(data={'a': 5}, errors=None)

schema = Schema({'a': 5})
schema({'a': 6})
# SchemaResult(data=None, errors={'a': 'bad value: value error, '
#                                      'expected 5 but found 6'})
```

Types

Validate against one (by using a single type, e.g. `str`) or more (by using a tuple of types, e.g. `(str, int, float)`) types:

```
from schemable import Schema

schema = Schema(str)
schema('a')
# SchemaResult(data='a', errors=None)

schema = Schema(int)
schema('5')
# SchemaResult(data=None, errors='type error, expected int but found str')

schema = Schema((int, str))
schema('5')
# SchemaResult(data='5', errors=None)
```

Predicates

Predicates are simply callables that either return truthy or `None` (on successful validation) or falsey or raise an exception (on failed validation):

```
from schemable import Schema

schema = Schema(lambda x: x > 5)
schema(6)
# SchemaResult(data=6, errors=None)

schema = Schema(lambda x: x > 5)
schema(4)
# SchemaResult(data=None, errors='<lambda>(4) should evaluate to True')

def gt_5(x): return x > 5
schema = Schema(gt_5)
schema(4)
# SchemaResult(data=None, errors='gt_5(4) should evaluate to True')
```

All

The *All* helper is used to validate against multiple predicates where all predicates must pass:

```
from schemable import Schema, All

def lt_10(x): return x < 10
def is_odd(x): return x % 2 == 1

schema = Schema(All(lt_10, is_odd))
schema(5)
# SchemaResult(data=5, errors=None)

schema = Schema(All(lt_10, is_odd))
schema(6)
# SchemaResult(data=None, errors='is_odd(6) should evaluate to True')
```

Any

The *Any* helper is used to validate against multiple predicates where at least one predicate must pass:

```
from schemable import Schema, Any

def is_float(x): return isinstance(x, float)
def is_int(x): return isinstance(x, int)

schema = Schema(Any(is_float, is_int))
schema(5)
# SchemaResult(data=5, errors=None)

schema = Schema(Any(is_float, is_int))
schema(5.2)
# SchemaResult(data=5.2, errors=None)

schema = Schema(Any(is_float, is_int))
```

(continues on next page)

(continued from previous page)

```
schema('a')
# SchemaResult(data=None, errors="is_int('a') should evaluate to True"))
```

Lists

List validation is primarily used to validate each item in a list against a schema while also checking that the parent object is, in fact, a `list`.

```
schema = Schema([str])

schema(['a', 'b', 'c'])
# SchemaResult(
#     data=['a', 'b', 'c'],
#     errors={})

schema(['a', 'b', 'c', 3])
# SchemaResult(
#     data=['a', 'b', 'c'],
#     errors={3: 'bad value: type error, expected str but found int'})

schema = Schema([(int, float)])
schema([1, 2.5, '3'])
# SchemaResult(
#     data=[1, 2.5],
#     errors={2: 'bad value: type error, expected float or int but found str'})
```

Dictionaries

Dictionary validation is one of the primary methods for creating schemas for validating things like JSON APIs, deserialized dictionaries, configuration objects, or any dict or dict-like object. These schemas are nestable and can be defined using dictionaries or lists or even other *Schema* instances defined elsewhere (i.e. *Schema* instances are reusable as part of a larger *Schema*).

```
from schemable import Schema, Optional

schema = Schema({
    'a': str,
    'b': int,
    Optional('c'): dict,
    'd': [{
        'e': str,
        'f': bool,
        'g': {
            'h': (int, float),
            'i': (int, bool)
        }
    }]
})

schema({
    'a': 'j',
    'b': 1,
    'd': [
```

(continues on next page)

(continued from previous page)

```
        {'e': 'k', 'f': True, 'g': {'h': 1, 'i': False}},
        {'e': 'l', 'f': False, 'g': {'h': 1.5, 'i': 0}},
    ]
})
# SchemaResult(
#     data={'a': 'j',
#           'b': 1,
#           'd': [{'e': 'k', 'f': True, 'g': {'h': 1, 'i': False}},
#                 {'e': 'l', 'f': False, 'g': {'h': 1.5, 'i': 0}}]},
#     errors={})

schema({
    'a': 'j',
    'b': 1,
    'c': {'x': 1, 'y': 2},
    'd': [
        {'e': 'k', 'f': True, 'g': {'h': 1, 'i': False}},
        {'e': 'l', 'f': False, 'g': {'h': 1.5, 'i': 0}},
    ]
})
# SchemaResult(
#     data={'a': 'j',
#           'b': 1,
#           'c': {'x': 1, 'y': 2},
#           'd': [{'e': 'k', 'f': True, 'g': {'h': 1, 'i': False}},
#                 {'e': 'l', 'f': False, 'g': {'h': 1.5, 'i': 0}}]},
#     errors={})

schema({
    'a': 'j',
    'b': 1,
    'c': [1, 2, 3],
    'd': [
        {'e': 'k', 'f': True, 'g': {'h': False, 'i': False}},
        {'e': 10, 'f': False, 'g': {'h': 1.5, 'i': 1.5}},
    ]
})
# SchemaResult(
#     data={'a': 'j',
#           'b': 1,
#           'd': [{'e': 'k', 'f': True, 'g': {'i': False}},
#                 {'f': False, 'g': {'h': 1.5}}]},
#     errors={'c': 'bad value: type error, expected dict but found list',
#             'd': {0: {'g': {'h': 'bad value: type error, expected float '
#                                  'or int but found bool'}},
#                   1: {'e': 'bad value: type error, expected str but '
#                            'found int',
#                       'g': {'i': 'bad value: type error, expected bool '
#                                  'or int but found float'}}}})
```

By default all keys are required unless wrapped with *Optional*. This includes key types like `Schema({str: str})` where that at least one data key must match all non-optional schema keys:

```
from schema import Schema, Optional

# Fails due to missing at least one integer key.
```

(continues on next page)

(continued from previous page)

```
Schema({str: str, int: int})({'a': 'b'})
# SchemaResult(data={'a': 'b'}, errors={<class 'int'>: 'missing required key'})

# But this passes.
Schema({str: str, Optional(int): int})({'a': 'b'})
# SchemaResult(data={'a': 'b'}, errors={})
```

Optional keys can define a default using the `default` argument:

```
from schemable import Schema, Optional

schema = Schema({
    Optional('a'): str,
    Optional('b', default=5): str,
    Optional('c', default=dict): str
})

schema({})
# SchemaResult(data={'b': 5, 'c': {}}, errors={})
```

TIP: For mutable defaults, always use a callable that returns a new instance. For example, for `{}` use `dict`, for `[]` use `list`, etc. This prevents bugs where the same object is used for separate schema results that results in changes to one affecting all the others.

When determining how to handle extra keys (i.e. keys in the data but not matched in the schema), there are three modes:

- `ALLOW_EXTRA`: Any extra keys are passed to *`SchemaResult`* as-is.
- `DENY_EXTRA`: Any extra keys result in failed validation.
- `IGNORE_EXTRA` (the default): All extra keys are ignored and won't appear in *`SchemaResult`*.

The "extra" mode is set via `Schema(..., extra=ALLOW_EXTRA|DENY_EXTRA|IGNORE_EXTRA)`:

```
from schemable import ALLOW_EXTRA, DENY_EXTRA, IGNORE_EXTRA, Schema, Optional

Schema({int: int})({1: 1, 'a': 'a'})
# SchemaResult(data={1: 1}, errors={})

# Same as above.
Schema({int: int}, extra=IGNORE_EXTRA)({1: 1, 'a': 'a'})
# SchemaResult(data={1: 1}, errors={})

Schema({int: int}, extra=ALLOW_EXTRA)({1: 1, 'a': 'a'})
# SchemaResult(data={1: 1, 'a': 'a'}, errors={})

Schema({int: int}, extra=DENY_EXTRA)({1: 1, 'a': 'a'})
# SchemaResult(data={1: 1}, errors={'a': "bad key: not in [<class 'int'>]"})
```

For some schemas, data keys may logically match multiple schema keys (e.g. `{'a': int, str: str, (str, int): bool}`). However, value-based key schemas are treated differently than type-based or other key schemas when it comes to validation resolution. The value-based key schemas will take precedence over all others and will essentially "swallow" a key-value pair so that the value-based key schema must pass (while other key-schemas are ignored for a particular data key):

```
from schemable import Schema

```

(continues on next page)

(continued from previous page)

```
schema = Schema({
    'a': int,
    str: str,
})

# Value-based key schema takes precedence
schema({'a': 'foo', 'x': 'y'})
# SchemaResult(
#     data={'x': 'y'},
#     errors={'a': 'bad value: type error, expected int but found str'})

schema({'a': 1, 'x': 'y'})
# SchemaResult(data={'a': 1, 'x': 'y'}, errors={})
```

For non-value-based key schemas (in the absence of a value-based key match) *all* key schemas will be checked. Each matching key schema's value schema will then be used with *Any* when evaluating the data value. As long as at least one of the data-value schemas match, the data key-value will validate. However, be aware that multiple matching key schemas likely indicates that the schema can be rewritten so that keys will only match a single key schema. Generally, this is preferrable since it makes the schema more deterministic and probably more "correct".

```
from schemable import Schema

item = {'a': 1, 'x': 'y', 1: False, 2.5: 10.0, 'b': True}

# Instead of this.
Schema({
    'a': int,
    str: str,
    (str, int): bool,
    (int, float): float
})(item)
# SchemaResult(data={'a': 1, 'x': 'y', 1: False, 2.5: 10.0, 'b': True}, errors={})

# Rewrite the schema to this.
Schema({
    'a': int,
    str: (str, bool),
    int: (bool, float),
    float: float
})(item)
# SchemaResult(data={'a': 1, 'x': 'y', 1: False, 2.5: 10.0, 'b': True}, errors={})
```

4.2.2 Transformation

In addition to validation, Schemable can transform data into computed values. Transformations can also be combined with validation using *All* to ensure data is only transformed after passing validation.

```
from schemable import Schema, All, As

# Validated that object is an integer or float.
# Then transform it to a float.
schema = Schema(All((int, float), As(float)))

schema(1)
```

(continues on next page)

(continued from previous page)

```
# SchemaResult(data=1.0, errors=None)

schema('a')
# SchemaResult(data=None, errors='type error, expected float or int but found str')
```

Select

The *Select* helper is used to "select" data from a source mapping (typically just a dictionary) and optionally transform it. The main usage patterns are:

- `Select(<callable>)`: Select and modify the source using `<callable>` as in `mycallable(source)`. Typically use-case is to return computed data that uses one or more source fields.
- `Select('<field>')`: Select `'<field>'` from source and return it as-is as in `source['field']`. Typically use-case is to alias a source field.
- `Select('<field>', <callable>)`: Select `'<field>'` from source and modify it using `<callable>` as in `mycallable(source['field'])`. This is actually equivalent to `All(Select('field'), mycallable)` but provides a terser syntax.

```
from schemable import Schema, Select

schema = Schema({
    'items': [str],
    'total_items': Select('items', len),
    'user_settings': Select('userSettings'),
    'full_name': Select(lambda d: '{} {}'.format(d['firstName'], d['lastName']))
})

schema({
    'items': ['a', 'b', 'c'],
    'userSettings': {},
    'firstName': 'Alice',
    'lastName': 'Smith'
})
# SchemaResult(
#     data={'total_items': 3,
#           'user_settings': {},
#           'full_name': 'Alice Smith',
#           'items': ['a', 'b', 'c']},
#     errors={})
```

As

The *As* helper is used to transform data into another value using a callable. For dictionary schemas, this helper can transform the source value (unlike *Select* which can transform any part of the source). It is equivalent to `{'a': Select('a', func)}` but provides a terser syntax.

```
from schemable import Schema, All, As

schema = Schema({
    'a': As(int),
    'b': All(int, As(float))
```

(continues on next page)

(continued from previous page)

```
})

schema({'a': '5', 'b': 3})
# SchemaResult(data={'a': 5, 'b': 3.0}, errors={})

schema({'a': '5', 'b': 3.5})
# SchemaResult(
#     data={'a': 5},
#     errors={'b': 'bad value: type error, expected int but found float'})

schema({'a': 'x', 'b': 3})
# SchemaResult(
#     data={'b': 3.0},
#     errors={'a': "bad value: int('x') should not raise an exception: "
#                  "invalid literal for int() with base 10: 'x'"})
```

When used with *All*, each argument to *All* will be evaulated in series and composed so that multiple usage of *As* will simply transform the previous result.

```
schema = Schema(All(As(int), As(float)))
schema(1.5)
# SchemaResult(data=1.0, errors=None)
```

Use

The *Use* helper returns either a constant value or the result of a callable called without any arguments.

```
from schemable import Schema, Use
from datetime import datetime

schema = Schema({
    'api_version': Use('v1'),
    'timestamp': Use(datetime.now)
})

schema({})
# SchemaResult(
#     data={'api_version': 'v1',
#           'timestamp': datetime.datetime(2018, 7, 28, 21, 47, 16, 365280)},
#     errors={})
```

4.2.3 Related Libraries

Schemable borrows featues from several other schema libraries:

- schema
- voluptuous
- marshmallow

However, the main difference with Schemable is that it provides an interface similar to `schema` and `voluptuous` (i.e. simple object schema declartions using dicts/lists instead of classes) but supports partial data loading like `marshmallow`. But unlike `marshamallow`, there is no concept of loading/dumping or deserialization/serialization; there's just validation, transformation, and parsing (the de/serialization is left up to the developer).

4.3 API Reference

4.3.1 Schema

The schema module.

class schemable.schema.**Schema**(*spec*, *strict=False*, *extra=None*)
The primary schema class that defines the validation and loading specification of a schema.

This class is used to create a top-level schema object that can be called on input data to validation and load it according to the specification.

Parameters

- **spec** (*object*) – Schema specification.
- **strict** (*bool, optional*) – Whether to evaluate schema in strict mode that will raise an exception on failed validation. Defaults to `False`.
- **extra** (*bool|None, optional*) – Sets the extra keys policy when validating *Dict* schemas. Defaults to `IGNORE_EXTRA`.

class schemable.base.**SchemaResult**
The result returned from schema evaluation.

data
object – Parsed data that passed schema validation.

errors
object – Schema errors as `None`, `dict`, or `str` depending on whethere there were any errors and what kind of schema was defined.

exception schemable.base.**SchemaError**(*message*, *errors*, *data*, *original_data*)
Exception raised when schema validation fails during strict schema mode.

message
str – Generic error message.

errors
str|dict – Schema validation error string or dictionary.

data
list|dict|None – Partially parsed data or `None`.

original_data
object – Original data being validated.

4.3.2 Validators

The validators module.

class schemable.validators.**All**(**specs*)
Schema helper that validates against a list of schemas where all schems must validate.

Parameters ***specs** (*object*) – Schema specifications to validate against.

class schemable.validators.**Any**(**specs*)
Schema helper that validates against a list of schemas where at least one schema must validate.

Parameters ***specs** (*object*) – Schema specifications to validate against.

class schemable.validators.**Dict**(*spec*, *extra=None*)
Schema helper that validates against dict or dict-like objects.

Parameters

- **spec** (*dict*) – Dictionary containing schema specification to validate against.
- **extra** (*bool|None, optional*) – Sets the extra keys policy. Defaults to IGNORE_EXTRA.

class schemable.validators.**List**(*spec*)
Schema helper that validates against list objects.

Parameters **spec** (*list*) – List containing schema specification to validate each list item against.

class schemable.validators.**Optional**(*spec*, *default=<NotSet>*)
Schema helper used to mark a *Dict* key as optional.

Parameters

- **spec** (*object*) – *Dict* key schema specification.
- **default** (*object, optional*) – Default value or callable that returns a default to be used when a key isn't given.

class schemable.validators.**Type**(*spec*)
Schema helper that validates against types.

Parameters **spec** (*type|tuple[type]*) – A type or tuple or tuple of types to validate against.

class schemable.validators.**Validate**(*spec*)
Schema helper that validates against a callable.

Validation passes if the callable returns None or a truthy value. Validation fails if the callable raises and exception or returns a non-None falsey value.

Parameters **spec** (*callable*) – Callable to validate against.

class schemable.validators.**Value**(*spec*)
Schema helper that validates against value equality.

Parameters **spec** (*object*) – Value to compare to.

4.3.3 Transforms

The transforms module.

class schemable.transforms.**As**(*spec*)
Schema helper that modifies a parsed schema value using a callable.

Unlike *Validate* the return value from the callable will replace the parsed value. However, if an exception occurs, validation will fail.

Parameters **spec** (*callable*) – Callable that transforms a value.

class schemable.transforms.**Select**(*spec*, *iteratee=<NotSet>*)
Schema helper that selects and optionally modifies source data.

There are three ways to use:

1. Select('<field>'): Returns data['<field>'].
2. Select(<callable>): Passes source data to <callable> and returned value is the schema result.

3. `Select('<field>', <callable>)`: Passes `data['<field>']` to <callable> and returned value is the schema result.

Parameters

- **spec** (*str|callable*) – The string field name to select from the source or a callable that accepts the source as its only argument.
- **iteratee** (*callable, optional*) – A callable that modifies a source object field value. Is used when *spec* is a string that selects a field from the source and *iteratee* then modifies the field value.

class schemable.transforms.**Use**(*spec*)

Schema helper that returns a constant value or the return from a callable while ignoring the source data.

Parameters **spec** (*object*) – Any object or callable.

CHAPTER 5

Project Info

5.1 License

5.2 Versioning

This project follows Semantic Versioning with the following caveats:

- Only the public API (i.e. the objects imported into the schemable module) will maintain backwards compatibility between MINOR version bumps.
- Objects within any other parts of the library are not guaranteed to not break between MINOR version bumps.

With that in mind, it is recommended to only use or import objects from the main module, schemable.

5.3 Changelog

5.3.1 v0.5.0 (2018-08-17)

- Don't load partial data from a nested schema if it was created with `strict=True` (e.g. `Schema({'key': Schema({...}, strict=True)})`).

5.3.2 v0.4.1 (2018-08-14)

- Fix previous fix for case where schema results could have `data` or `errors` with schema classes as keys.
- Ensure that `Select('key', <iteratee>)` doesn't call `<iteratee>` if `'key'` was not found in the source data.

5.3.3 v0.4.0 (2018-08-14)

- Fix case where schema object with an `Optional(key)` would result in `SchemaResult.errors[Optional(key)]`. Ensure that `SchemaResult.errors[key]` is set instead.
- Ignore `KeyError` when using `Schema({'key': Select('other_key')})` when `'other_key'` isn't present in the source object. Return a missing key error instead.

5.3.4 v0.3.1 (2018-07-31)

- If a validate callable raises an exception, use its string representation as the schema error message. Previously, a custom error message stating that the callable should evaluate to true was used when validator returned falsey and when it raised an exception. That message is now only returned when the validator doesn't raise but returns falsey.

5.3.5 v0.3.0 (2018-07-27)

- Add schema helpers:
  - `Select`
  - `Use`
- Include execption class name in error message returned by `As`.
- Always return a `dict` when parsing from dictionary schemas instead of trying to use the source data's type as an initializer. (**breaking change**)

5.3.6 v0.2.0 (2018-07-25)

- Rename `Collection` to `List`. (**breaking change**)
- Rename `Object` to `Dict`. (**breaking change**)
- Allow `collections.abc.Mapping` objects to be valid `Dict` objects.
- Modify `Type` validation so that objects are only compared with `isinstance`.
- Improve docs.

5.3.7 v0.1.0 (2018-07-24)

- First release.

5.4 Authors

5.4.1 Lead

- Derrick Gilland, dgilland@gmail.com, dgilland@github

5.4.2 Contributors

None

5.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/dgilland/schemable.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" or "help wanted" is open to whoever wants to implement it.

Write Documentation

schemable could always use more documentation, whether as part of the official schemable docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/dgilland/schemable.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.5.2 Get Started!

Ready to contribute? Here's how to set up `schemable` for local development.

1. Fork the `schemable` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_username_here/schemable.git
```

3. Install Python dependencies into a virtualenv:

```
$ cd schemable
$ pip install -r requirements-dev.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass linting and all unit tests by testing with tox across all supported Python versions:

```
$ tox
```

6. Add yourself to `AUTHORS.rst`.
7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

5.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the README.rst.
3. The pull request should work for all versions Python that this project supports. Check https://travis-ci.org/dgilland/schemable/pull_requests and make sure that the all environments pass.

CHAPTER 6

Indices and Tables

- genindex
- modindex
- search

Python Module Index

S

Index