jsonschema Documentation

Release 2.6.0

Julian Berman

February 05, 2017

Contents

`jsonschema` is an implementation of JSON Schema for Python (supporting 2.7+ including Python 3).

```
>>> from jsonschema import validate

>>> # A sample schema, like what we'd get from json.load()
>>> schema = {
...     "type" : "object",
...     "properties" : {
...         "price" : {"type" : "number"},
...         "name" : {"type" : "string"},
...     },
... }

>>> # If no exception is raised by validate(), the instance is valid.
>>> validate({"name" : "Eggs", "price" : 34.99}, schema)

>>> validate(
...     {"name" : "Eggs", "price" : "Invalid"}, schema
... )
Traceback (most recent call last):
    ...
ValidationError: 'Invalid' is not of type 'number'
```

You can find further information (installation instructions, mailing list) as well as the source code and issue tracker on our GitHub page.

Contents:

CHAPTER 1

Schema Validation

1.1 The Basics

The simplest way to validate an instance under a given schema is to use the *validate()* function.

jsonschema.**validate**(*instance*, *schema*, *cls=None*, **args*, ***kwargs*)

Validate an instance under the given schema.

```
>>> validate([2, 3, 4], {"maxItems": 2})
Traceback (most recent call last):
    ...
ValidationError: [2, 3, 4] is too long
```

validate() will first verify that the provided schema is itself valid, since not doing so can lead to less obvious error messages and fail in less obvious or consistent ways. If you know you have a valid schema already or don't care, you might prefer using the *validate()* method directly on a specific validator (e.g. `Draft4Validator.validate()`).

Parameters

- **instance** – The instance to validate
- **schema** – The schema to validate with
- **cls** (*IValidator*) – The class that will be used to validate the instance.

If the `cls` argument is not provided, two things will happen in accordance with the specification. First, if the schema has a $schema property containing a known meta-schema [1] then the proper validator will be used. The specification recommends that all schemas contain $schema properties for this reason. If no $schema property is found, the default validator class is *Draft4Validator*.

Any other provided positional and keyword arguments will be passed on when instantiating the `cls`.

Raises

- `ValidationError` if the instance is invalid
- `SchemaError` if the schema itself is invalid

1.2 The Validator Interface

jsonschema defines an (informal) interface that all validator classes should adhere to.

[1] known by a validator registered with `validates()`

class jsonschema.**IValidator**(*schema*, *types=()*, *resolver=None*, *format_checker=None*)

Parameters

- **schema** (*dict*) – the schema that the validator object will validate with. It is assumed to be valid, and providing an invalid schema can lead to undefined behavior. See *IValidator.check_schema()* to validate a schema first.
- **types** (*dict or iterable of 2-tuples*) – Override or extend the list of known types when validating the type property. Should map strings (type names) to class objects that will be checked via isinstance(). See *Validating With Additional Types* for details.
- **resolver** – an instance of *RefResolver* that will be used to resolve $ref properties (JSON references). If unprovided, one will be created.
- **format_checker** – an instance of *FormatChecker* whose conforms() method will be called to check and see if instances conform to each format property present in the schema. If unprovided, no validation will be done for format.

DEFAULT_TYPES

The default mapping of JSON types to Python types used when validating type properties in JSON schemas.

META_SCHEMA

An object representing the validator's meta schema (the schema that describes valid schemas in the given version).

VALIDATORS

A mapping of validator names (strs) to functions that validate the validator property with that name. For more information see *Creating or Extending Validator Classes*.

schema

The schema that was passed in when initializing the object.

classmethod check_schema(*schema*)

Validate the given schema against the validator's *META_SCHEMA*.

Raises SchemaError if the schema is invalid

is_type(*instance*, *type*)

Check if the instance is of the given (JSON Schema) type.

Return type bool

Raises UnknownType if type is not a known type.

is_valid(*instance*)

Check if the instance is valid under the current *schema*.

Return type

bool

```
>>> schema = {"maxItems" : 2}
>>> Draft3Validator(schema).is_valid([2, 3, 4])
False
```

iter_errors(*instance*)

Lazily yield each of the validation errors in the given instance.

Return type

an iterable of ValidationErrors

```
>>> schema = {
...     "type" : "array",
...     "items" : {"enum" : [1, 2, 3]},
...     "maxItems" : 2,
... }
>>> v = Draft3Validator(schema)
>>> for error in sorted(v.iter_errors([2, 3, 4]), key=str):
...     print(error.message)
4 is not one of [1, 2, 3]
[2, 3, 4] is too long
```

validate(*instance*)

Check if the instance is valid under the current *schema*.

Raises ValidationError if the instance is invalid

```
>>> schema = {"maxItems" : 2}
>>> Draft3Validator(schema).validate([2, 3, 4])
Traceback (most recent call last):
    ...
ValidationError: [2, 3, 4] is too long
```

All of the *versioned validators* that are included with *jsonschema* adhere to the interface, and implementors of validator classes that extend or complement the ones included should adhere to it as well. For more information see *Creating or Extending Validator Classes*.

1.2.1 Validating With Additional Types

Occasionally it can be useful to provide additional or alternate types when validating the JSON Schema's type property. Validators allow this by taking a types argument on construction that specifies additional types, or which can be used to specify a different set of Python types to map to a given JSON type.

jsonschema tries to strike a balance between performance in the common case and generality. For instance, JSON Schema defines a number type, which can be validated with a schema such as {"type" : "number"}. By default, this will accept instances of Python numbers.Number. This includes in particular ints and floats, along with decimal.Decimal objects, complex numbers etc. For integer and object, however, rather than checking for numbers.Integral and collections.abc.Mapping, *jsonschema* simply checks for int and dict, since the more general instance checks can introduce significant slowdown, especially given how common validating these types are.

If you *do* want the generality, or just want to add a few specific additional types as being acceptible for a validator object, *IValidator*s have a types argument that can be used to provide additional or new types.

```
class MyInteger(object):
    ...

Draft3Validator(
    schema={"type" : "number"},
    types={"number" : (numbers.Number, MyInteger)},
)
```

The list of default Python types for each JSON type is available on each validator object in the *IValidator.DEFAULT_TYPES* attribute. Note that you need to specify all types to match if you override one of the existing JSON types, so you may want to access the set of default types when specifying your additional type.

1.3 Versioned Validators

jsonschema ships with validator classes for various versions of the JSON Schema specification. For details on the methods and attributes that each validator class provides see the *IValidator* interface, which each included validator class implements.

class jsonschema.**Draft3Validator**(*schema*, *types=()*, *resolver=None*, *format_checker=None*)

class jsonschema.**Draft4Validator**(*schema*, *types=()*, *resolver=None*, *format_checker=None*)

For example, if you wanted to validate a schema you created against the Draft 4 meta-schema, you could use:

```
from jsonschema import Draft4Validator

schema = {
    "$schema": "http://json-schema.org/schema#",

    "type": "object",
    "properties": {
        "name": {"type": "string"},
        "email": {"type": "string"},
    },
    "required": ["email"]
}
Draft4Validator.check_schema(schema)
```

1.4 Validating Formats

JSON Schema defines the format property which can be used to check if primitive types (`strings`, `numbers`, `booleans`) conform to well-defined formats. By default, no validation is enforced, but optionally, validation can be enabled by hooking in a format-checking object into an *IValidator*.

```
>>> validate("localhost", {"format" : "hostname"})
>>> validate(
...     "-12", {"format" : "hostname"}, format_checker=FormatChecker(),
... )
Traceback (most recent call last):
    ...
ValidationError: "-12" is not a "hostname"
```

class jsonschema.**FormatChecker**(*formats=None*)

A `format` property checker.

JSON Schema does not mandate that the `format` property actually do any validation. If validation is desired however, instances of this class can be hooked into validators to enable format validation.

FormatChecker objects always return `True` when asked about formats that they do not know how to validate.

To check a custom format using a function that takes an instance and returns a `bool`, use the *FormatChecker.checks()* or *FormatChecker.cls_checks()* decorators.

Parameters **formats** (*iterable*) – The known formats to validate. This argument can be used to limit which formats will be used during validation.

checkers

A mapping of currently known formats to tuple of functions that validate them and errors that should be

caught. New checkers can be added and removed either per-instance or globally for all checkers using the *FormatChecker.checks()* or *FormatChecker.cls_checks()* decorators respectively.

classmethod cls_checks (*format*, *raises=()*)

Register a decorated function as *globally* validating a new format.

Any instance created after this function is called will pick up the supplied checker.

Parameters

- **format** (*str*) – the format that the decorated function will check
- **raises** (*Exception*) – the exception(s) raised by the decorated function when an invalid instance is found. The exception object will be accessible as the `ValidationError.cause` attribute of the resulting validation error.

check (*instance*, *format*)

Check whether the instance conforms to the given format.

Parameters

- **instance** (*any primitive type, i.e. str, number, bool*) – The instance to check
- **format** (*str*) – The format that instance should conform to

Raises `FormatError` if instance does not conform to `format`

checks (*format*, *raises=()*)

Register a decorated function as validating a new format.

Parameters

- **format** (*str*) – The format that the decorated function will check.
- **raises** (*Exception*) – The exception(s) raised by the decorated function when an invalid instance is found.

  The exception object will be accessible as the `ValidationError.cause` attribute of the resulting validation error.

conforms (*instance*, *format*)

Check whether the instance conforms to the given format.

Parameters

- **instance** (*any primitive type, i.e. str, number, bool*) – The instance to check
- **format** (*str*) – The format that instance should conform to

Returns Whether it conformed

Return type bool

There are a number of default checkers that *FormatChecker*s know how to validate. Their names can be viewed by inspecting the *FormatChecker.checkers* attribute. Certain checkers will only be available if an appropriate package is available for use. The available checkers, along with their requirement (if any,) are listed below.

Checker	Notes
hostname	
ipv4	
ipv6	OS must have `socket.inet_pton()` function
email	
uri	requires rfc3987
date-time	requires strict-rfc3339 [2]
date	
time	
regex	
color	requires webcolors

[2] For information on creating JSON schemas to validate your data, there is a good introduction to JSON Schema fundamentals underway at Understanding JSON Schema

CHAPTER 2

Handling Validation Errors

When an invalid instance is encountered, a *ValidationError* will be raised or returned, depending on which method or function is used.

exception jsonschema.exceptions.**ValidationError**(*message*, *validator=<unset>*, *path=()*, *cause=None*, *context=()*, *validator_value=<unset>*, *instance=<unset>*, *schema=<unset>*, *schema_path=()*, *parent=None*)

The instance didn't properly validate under the provided schema.

The information carried by an error roughly breaks down into:

What Happened	Why Did It Happen	What Was Being Validated
message	*context* *cause*	*instance* *path* *schema* *schema_path* *validator* *validator_value*

message
A human readable message explaining the error.

validator
The name of the failed validator.

validator_value
The value for the failed validator in the schema.

schema
The full schema that this error came from. This is potentially a subschema from within the schema that was passed in originally, or even an entirely different schema if a $ref was followed.

relative_schema_path
A collections.deque containing the path to the failed validator within the schema.

absolute_schema_path
A collections.deque containing the path to the failed validator within the schema, but always relative to the *original* schema as opposed to any subschema (i.e. the one originally passed into a validator class, *not* *schema*).

schema_path
Same as *relative_schema_path*.

relative_path
A `collections.deque` containing the path to the offending element within the instance. The deque can be empty if the error happened at the root of the instance.

absolute_path
A `collections.deque` containing the path to the offending element within the instance. The absolute path is always relative to the *original* instance that was validated (i.e. the one passed into a validation method, *not* `instance`). The deque can be empty if the error happened at the root of the instance.

path
Same as *relative_path*.

instance
The instance that was being validated. This will differ from the instance originally passed into `validate()` if the validator object was in the process of validating a (possibly nested) element within the top-level instance. The path within the top-level instance (i.e. *ValidationError.path*) could be used to find this object, but it is provided for convenience.

context
If the error was caused by errors in subschemas, the list of errors from the subschemas will be available on this property. The *schema_path* and *path* of these errors will be relative to the parent error.

cause
If the error was caused by a *non*-validation error, the exception object will be here. Currently this is only used for the exception raised by a failed format checker in `FormatChecker.check()`.

parent
A validation error which this error is the *context* of. `None` if there wasn't one.

In case an invalid schema itself is encountered, a *SchemaError* is raised.

exception `jsonschema.exceptions.`**SchemaError**(*message*, *validator=<unset>*, *path=()*, *cause=None*, *context=()*, *validator_value=<unset>*, *instance=<unset>*, *schema=<unset>*, *schema_path=()*, *parent=None*)

The provided schema is malformed.

The same attributes are present as for *ValidationError*s.

These attributes can be clarified with a short example:

```
schema = {
    "items": {
        "anyOf": [
            {"type": "string", "maxLength": 2},
            {"type": "integer", "minimum": 5}
        ]
    }
}
instance = [{}, 3, "foo"]
v = Draft4Validator(schema)
errors = sorted(v.iter_errors(instance), key=lambda e: e.path)
```

The error messages in this situation are not very helpful on their own.

```
for error in errors:
    print(error.message)
```

outputs:

```
{} is not valid under any of the given schemas
3 is not valid under any of the given schemas
'foo' is not valid under any of the given schemas
```

If we look at *path* on each of the errors, we can find out which elements in the instance correspond to each of the errors. In this example, *path* will have only one element, which will be the index in our list.

```
for error in errors:
    print(list(error.path))
```

```
[0]
[1]
[2]
```

Since our schema contained nested subschemas, it can be helpful to look at the specific part of the instance and subschema that caused each of the errors. This can be seen with the *instance* and *schema* attributes.

With validators like anyOf, the *context* attribute can be used to see the sub-errors which caused the failure. Since these errors actually came from two separate subschemas, it can be helpful to look at the *schema_path* attribute as well to see where exactly in the schema each of these errors come from. In the case of sub-errors from the *context* attribute, this path will be relative to the *schema_path* of the parent error.

```
for error in errors:
    for suberror in sorted(error.context, key=lambda e: e.schema_path):
        print(list(suberror.schema_path), suberror.message, sep=", ")
```

```
[0, 'type'], {} is not of type 'string'
[1, 'type'], {} is not of type 'integer'
[0, 'type'], 3 is not of type 'string'
[1, 'minimum'], 3 is less than the minimum of 5
[0, 'maxLength'], 'foo' is too long
[1, 'type'], 'foo' is not of type 'integer'
```

The string representation of an error combines some of these attributes for easier debugging.

```
print(errors[1])
```

```
3 is not valid under any of the given schemas

Failed validating 'anyOf' in schema['items']:
    {'anyOf': [{'maxLength': 2, 'type': 'string'},
               {'minimum': 5, 'type': 'integer'}]}

On instance[1]:
    3
```

2.1 ErrorTrees

If you want to programmatically be able to query which properties or validators failed when validating a given instance, you probably will want to do so using `ErrorTree` objects.

class `jsonschema.validators.`**`ErrorTree`**(*errors=()*)

ErrorTrees make it easier to check which validations failed.

`errors`

The mapping of validator names to the error objects (usually *ValidationError*s) at this level of the tree.

__contains__(*index*)
Check whether `instance[index]` has any errors.

__getitem__(*index*)
Retrieve the child tree one level down at the given `index`.

If the index is not in the instance that this tree corresponds to and is not known by this tree, whatever error would be raised by `instance.__getitem__` will be propagated (usually this is some subclass of `LookupError`.

__iter__()
Iterate (non-recursively) over the indices in the instance with errors.

__len__()
Same as `total_errors`.

total_errors
The total number of errors in the entire tree, including children.

Consider the following example:

```
schema = {
    "type" : "array",
    "items" : {"type" : "number", "enum" : [1, 2, 3]},
    "minItems" : 3,
}
instance = ["spam", 2]
```

For clarity's sake, the given instance has three errors under this schema:

```
v = Draft3Validator(schema)
for error in sorted(v.iter_errors(["spam", 2]), key=str):
    print(error.message)
```

```
'spam' is not of type 'number'
'spam' is not one of [1, 2, 3]
['spam', 2] is too short
```

Let's construct an `ErrorTree` so that we can query the errors a bit more easily than by just iterating over the error objects.

```
tree = ErrorTree(v.iter_errors(instance))
```

As you can see, `ErrorTree` takes an iterable of *`ValidationError`*s when constructing a tree so you can directly pass it the return value of a validator object's `iter_errors` method.

`ErrorTree`s support a number of useful operations. The first one we might want to perform is to check whether a given element in our instance failed validation. We do so using the `in` operator:

```
>>> 0 in tree
True

>>> 1 in tree
False
```

The interpretation here is that the 0th index into the instance (`"spam"`) did have an error (in fact it had 2), while the 1th index (`2`) did not (i.e. it was valid).

If we want to see which errors a child had, we index into the tree and look at the *`errors`* attribute.

```
>>> sorted(tree[0].errors)
['enum', 'type']
```

Here we see that the enum and type validators failed for index 0. In fact *errors* is a dict, whose values are the *ValidationError*s, so we can get at those directly if we want them.

```
>>> print(tree[0].errors["type"].message)
'spam' is not of type 'number'
```

Of course this means that if we want to know if a given named validator failed for a given index, we check for its presence in *errors*:

```
>>> "enum" in tree[0].errors
True

>>> "minimum" in tree[0].errors
False
```

Finally, if you were paying close enough attention, you'll notice that we haven't seen our minItems error appear anywhere yet. This is because minItems is an error that applies globally to the instance itself. So it appears in the root node of the tree.

```
>>> "minItems" in tree.errors
True
```

That's all you need to know to use error trees.

To summarize, each tree contains child trees that can be accessed by indexing the tree to get the corresponding child tree for a given index into the instance. Each tree and child has a *errors* attribute, a dict, that maps the failed validator name to the corresponding validation error.

2.2 best_match and relevance

The *best_match()* function is a simple but useful function for attempting to guess the most relevant error in a given bunch.

```
>>> from jsonschema import Draft4Validator
>>> from jsonschema.exceptions import best_match

>>> schema = {
...     "type": "array",
...     "minItems": 3,
... }
>>> print(best_match(Draft4Validator(schema).iter_errors(11)).message)
11 is not of type 'array'
```

jsonschema.exceptions.**best_match**(*errors*, *key=<function relevance>*)

Try to find an error that appears to be the best match among given errors.

In general, errors that are higher up in the instance (i.e. for which *ValidationError.path* is shorter) are considered better matches, since they indicate "more" is wrong with the instance.

If the resulting match is either oneOf or anyOf, the *opposite* assumption is made – i.e. the deepest error is picked, since these validators only need to match once, and any other errors may not be relevant.

Parameters

- **errors** (*iterable*) – the errors to select from. Do not provide a mixture of errors from different validation attempts (i.e. from different instances or schemas), since it won't produce sensical output.

- **key** (*callable*) – the key to use when sorting errors. See *relevance* and transitively *by_relevance()* for more details (the default is to sort with the defaults of that function). Changing the default is only useful if you want to change the function that rates errors but still want the error context decension done by this function.

Returns the best matching error, or `None` if the iterable was empty

---

Note: This function is a heuristic. Its return value may change for a given set of inputs from version to version if better heuristics are added.

`jsonschema.exceptions.`**`relevance`**(*validation_error*)

A key function that sorts errors based on heuristic relevance.

If you want to sort a bunch of errors entirely, you can use this function to do so. Using this function as a key to e.g. `sorted()` or `max()` will cause more relevant errors to be considered greater than less relevant ones.

Within the different validators that can fail, this function considers anyOf and oneOf to be *weak* validation errors, and will sort them lower than other validators at the same level in the instance.

If you want to change the set of weak [or strong] validators you can create a custom version of this function with *by_relevance()* and provide a different set of each.

```
>>> schema = {
...     "properties": {
...         "name": {"type": "string"},
...         "phones": {
...             "properties": {
...                 "home": {"type": "string"}
...             },
...         },
...     },
... }
>>> instance = {"name": 123, "phones": {"home": [123]}}
>>> errors = Draft4Validator(schema).iter_errors(instance)
>>> [
...     e.path[-1]
...     for e in sorted(errors, key=exceptions.relevance)
... ]
['home', 'name']
```

`jsonschema.exceptions.`**`by_relevance`**(*weak=frozenset(['oneOf', 'anyOf'])*, *strong=frozenset([])*)

Create a key function that can be used to sort errors by relevance.

Parameters

- **weak** (*set*) – a collection of validator names to consider to be "weak". If there are two errors at the same level of the instance and one is in the set of weak validator names, the other error will take priority. By default, anyOf and oneOf are considered weak validators and will be superceded by other same-level validation errors.
- **strong** (*set*) – a collection of validator names to consider to be "strong"

CHAPTER 3

Resolving JSON References

class jsonschema.**RefResolver**(*base_uri*, *referrer*, *store=()*, *cache_remote=True*, *handlers=()*, *urljoin_cache=None*, *remote_cache=None*)

Resolve JSON References.

Parameters

- **base_uri** (*str*) – The URI of the referring document
- **referrer** – The actual referring document
- **store** (*dict*) – A mapping from URIs to documents to cache
- **cache_remote** (*bool*) – Whether remote refs should be cached after first resolution
- **handlers** (*dict*) – A mapping from URI schemes to functions that should be used to retrieve them
- **urljoin_cache** (*functools.lru_cache*) – A cache that will be used for caching the results of joining the resolution scope to subscopes.
- **remote_cache** (*functools.lru_cache*) – A cache that will be used for caching the results of resolved remote URLs.

classmethod from_schema(*schema*, **args*, ***kwargs*)

Construct a resolver from a JSON schema object.

Parameters **schema** – the referring schema

Returns *RefResolver*

resolve_fragment(*document*, *fragment*)

Resolve a `fragment` within the referenced `document`.

Parameters

- **document** – The referrant document
- **fragment** (*str*) – a URI fragment to resolve within it

resolve_remote(*uri*)

Resolve a remote `uri`.

If called directly, does not check the store first, but after retrieving the document at the specified URI it will be saved in the store if `cache_remote` is True.

Note: If the requests library is present, `jsonschema` will use it to request the remote `uri`, so that the correct encoding is detected and used.

If it isn't, or if the scheme of the `uri` is not `http` or `https`, UTF-8 is assumed.

Parameters **uri** (*str*) – The URI to resolve

Returns The retrieved document

resolving(**args*, ***kwds*)
Context manager which resolves a JSON `ref` and enters the resolution scope of this ref.

Parameters **ref** (*str*) – The reference to resolve

exception jsonschema.**RefResolutionError**
A JSON reference failed to resolve.

CHAPTER 4

Creating or Extending Validator Classes

jsonschema.validators.**create**(*meta_schema*, *validators=()*, *version=None*, *default_types=None*)
Create a new validator class.

Parameters

- **meta_schema** (*dict*) – the meta schema for the new validator class
- **validators** (*dict*) – a mapping from names to callables, where each callable will validate the schema property with the given name.

  Each callable should take 4 arguments:

  1. a validator instance,
  2. the value of the property being validated within the instance
  3. the instance
  4. the schema
- **version** (*str*) – an identifier for the version that this validator class will validate. If provided, the returned validator class will have its __name__ set to include the version, and also will have *validates()* automatically called for the given version.
- **default_types** (*dict*) – a default mapping to use for instances of the validator class when mapping between JSON types to Python types. The default for this argument is probably fine. Instances can still have their types customized on a per-instance basis.

Returns a new *jsonschema.IValidator* class

jsonschema.validators.**extend**(*validator*, *validators*, *version=None*)
Create a new validator class by extending an existing one.

Parameters

- **validator** (jsonschema.IValidator) – an existing validator class
- **validators** (*dict*) – a mapping of new validator callables to extend with, whose structure is as in *create()*.

  Note: Any validator callables with the same name as an existing one will (silently) replace the old validator callable entirely, effectively overriding any validation done in the "parent" validator class.

If you wish to instead extend the behavior of a parent's validator callable, delegate and call it directly in the new validator function by retrieving it using `OldValidator.VALIDATORS["validator_name"]`.

- **version** (*str*) – a version for the new validator class

Returns a new *jsonschema.IValidator* class

Note: Meta Schemas

The new validator class will have its parent's meta schema.

If you wish to change or extend the meta schema in the new validator class, modify `META_SCHEMA` directly on the returned class. Note that no implicit copying is done, so a copy should likely be made before modifying it, in order to not affect the old validator.

`jsonschema.validators.`**validator_for**(*schema*, *default=<unset>*)

Retrieve the validator class appropriate for validating the given schema.

Uses the $schema property that should be present in the given schema to look up the appropriate validator class.

Parameters

- **schema** – the schema to look at
- **default** – the default to return if the appropriate validator class cannot be determined. If unprovided, the default is to return `Draft4Validator`

`jsonschema.validators.`**validates**(*version*)

Register the decorated validator for a `version` of the specification.

Registered validators and their meta schemas will be considered when parsing `$schema` properties' URIs.

Parameters **version** (*str*) – An identifier to use as the version's name

Returns a class decorator to decorate the validator with the version

Return type callable

4.1 Creating Validation Errors

Any validating function that validates against a subschema should call `ValidatorMixin.descend()`, rather than `ValidatorMixin.iter_errors()`. If it recurses into the instance, or schema, it should pass one or both of the `path` or `schema_path` arguments to `ValidatorMixin.descend()` in order to properly maintain where in the instance or schema respectively the error occurred.

CHAPTER 5

Frequently Asked Questions

5.1 Why doesn't my schema that has a default property actually set the default on my instance?

The basic answer is that the specification does not require that default actually do anything.

For an inkling as to *why* it doesn't actually do anything, consider that none of the other validators modify the instance either. More importantly, having default modify the instance can produce quite peculiar things. It's perfectly valid (and perhaps even useful) to have a default that is not valid under the schema it lives in! So an instance modified by the default would pass validation the first time, but fail the second!

Still, filling in defaults is a thing that is useful. *jsonschema* allows you to define your own validator classes and callables, so you can easily create a `IValidator` that does do default setting. Here's some code to get you started. (In this code, we add the default properties to each object *before* the properties are validated, so the default values themselves will need to be valid under the schema.)

```
    from jsonschema import Draft4Validator, validators


    def extend_with_default(validator_class):
        validate_properties = validator_class.VALIDATORS["properties"]

        def set_defaults(validator, properties, instance, schema):
            for property, subschema in properties.iteritems():
                if "default" in subschema:
                    instance.setdefault(property, subschema["default"])

            for error in validate_properties(
                validator, properties, instance, schema,
            ):
                yield error

        return validators.extend(
            validator_class, {"properties" : set_defaults},
        )


    DefaultValidatingDraft4Validator = extend_with_default(Draft4Validator)


    # Example usage:
    obj = {}
```

```
    schema = {'properties': {'foo': {'default': 'bar'}}}
    # Note jsonschem.validate(obj, schema, cls=DefaultValidatingDraft4Validator)
    # will not work because the metaschema contains `default` directives.
    DefaultValidatingDraft4Validator(schema).validate(obj)
    assert obj == {'foo': 'bar'}
```

See the above-linked document for more info on how this works, but basically, it just extends the properties validator on a `Draft4Validator` to then go ahead and update all the defaults.

Note: If you're interested in a more interesting solution to a larger class of these types of transformations, keep an eye on Seep, which is an experimental data transformation and extraction library written on top of *jsonschema*.

Hint: The above code can provide default values for an entire object and all of its properties, but only if your schema provides a default value for the object itself, like so:

```
schema = {
    "type": "object",
    "properties": {
        "outer-object": {
            "type": "object",
            "properties" : {
                "inner-object": {
                    "type": "string",
                    "default": "INNER-DEFAULT"
                }
            },
            "default": {} # <-- MUST PROVIDE DEFAULT OBJECT
        }
    }
}

obj = {}
DefaultValidatingDraft4Validator(schema).validate(obj)
assert obj == {'outer-object': {'inner-object': 'INNER-DEFAULT'}}
```

...but if you don't provide a default value for your object, then it won't be instantiated at all, much less populated with default properties.

```
del schema["properties"]["outer-object"]["default"]
obj2 = {}
DefaultValidatingDraft4Validator(schema).validate(obj2)
assert obj2 == {} # whoops
```

5.2 How do jsonschema version numbers work?

`jsonschema` tries to follow the Semantic Versioning specification.

This means broadly that no backwards-incompatible changes should be made in minor releases (and certainly not in dot releases).

The full picture requires defining what constitutes a backwards-incompatible change.

The following are simple examples of things considered public API, and therefore should *not* be changed without bumping a major version number:

- module names and contents, when not marked private by Python convention (a single leading underscore)
- function and object signature (parameter order and name)

The following are *not* considered public API and may change without notice:

- the exact wording and contents of error messages; typical reasons to do this seem to involve unit tests. API users are encouraged to use the extensive introspection provided in `ValidationError`s instead to make meaningful assertions about what failed.
- the order in which validation errors are returned or raised
- the `compat.py` module, which is for internal compatibility use
- anything marked private

With the exception of the last two of those, flippant changes are avoided, but changes can and will be made if there is improvement to be had. Feel free to open an issue ticket if there is a specific issue or question worth raising.

CHAPTER 6

Indices and tables

- genindex
- search

j

Symbols

A

B

C

D

E

F

I

J

M

P

R

S

T

V