validictory Documentation

Release 1.0.1

James Turk

June 18, 2016

Contents

CHAPTER 1

Overview

validictory is a general purpose Python data validator that allows validation of arbitrary Python data structures.

Schema format is based on the JSON Schema proposal, so combined with `json` the library is also useful as a validator for JSON data.

Contains code derived from jsonschema by Ian Lewis and Ysuke Muraoka.

CHAPTER 2

Obtaining validictory

Source is available from GitHub.

The latest release is always available on PyPI and can be installed via pip.

Documentation lives at ReadTheDocs.

CHAPTER 3

Contents

3.1 Using validictory

Normal use of validictory is as simple as calling *validictory.validate()*, the only thing to learn is how to craft a schema.

3.1.1 Sample Usage

JSON documents and schema must first be loaded into a Python dictionary type before it can be validated.

Parsing a simple JSON document:

```
>>> import validictory

>>> validictory.validate("roast beef", {"type":"string"})
```

Parsing a more complex JSON document:

```
>>> import json
>>> import validictory

>>> data = json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
>>> schema = {
...    "type":"array",
...    "items":[
...      {"type":"string"},
...      {"type":"object",
...       "properties":{
...         "bar":{
...           "items":[
...             {"type":"string"},
...             {"type":"any"},
...             {"type":"number"},
...             {"type":"integer"}
...           ]
...         }
...       }
...     }
...    ]
... }
>>> validictory.validate(data,schema)
```

Catch ValueErrors to handle validation issues:

```
>>> import validictory

>>> try:
...     validictory.validate("short", {"type":"string","minLength":15})
... except ValueError, error:
...     print error
...
Length of value 'short' for field '_data' must be greater than or equal to 15
```

For more example usage of all schema options check out the tests within `validictory/tests`.

3.1.2 Schema Options

`type` Validate that an item in the data is of a particular type.

If a list of values is provided then any of the specified types will be accepted.

Provided value can be any combination of the following:

- `string` - str and unicode objects
- `integer` - ints
- `number` - ints and floats
- `boolean` - bools
- `object` - dicts
- `array` - lists and tuples
- `null` - None
- `any` - any type is acceptable

`properties` List of validators for properties of the object.

In essence each item in the provided dict for properties is a sub-schema applied against the property (if present) with the same name in the data.

```
# each key in the 'properties' option matches a key in the object that you are validating,
# and the value to each key in the 'properties' option is the schema to validate
# the value of the key in the JSON you are verifying.

data = json.loads(''' {"obj1": {"obj2": 12}}''' )

schema =
{
    "type": "object",
    "properties": {
        "obj1": {
            "type": "object",
            "properties": {
                "obj2": {
                    "type": "integer"
                }
            }
        }
    }
```

```
}
validictory.validate(data, schema)
```

patternProperties Define a set of patterns that validate against subschemas.

Similarly to how `properties` works, any properties in the data that have a name matching a particular pattern must validate against the provided sub-schema.

```
data = json.loads('''
    {
        "one": "hello",
        "two": "helloTwo",
        "thirtyThree": 12
    }''')

schema = {

    "type": "object",
    "properties": {
        "one": {
            "type": "string"
        },
        "two": {
            "type": "string"
        }
    },
    # each subkey of the 'patternProperties' option is a
    # regex, and the value is the schema to validate
    # all values whose keys match said regex.
    "patternProperties": {
        "^.+Three$": {
            "type": "number"
        }
    }

}
```

additionalProperties Schema for all additional properties not included in properties.

Can be `False` to disallow any additional properties not in `properties`, or can be a sub-schema that all properties not included in `properties` must match.

```
data = json.loads('''
    {
        "one": [12, 13],
        "two": "hello",
        "three": null,
        "four": null
    }''')

schema = {

    "type": "object",
    "properties": {

        "one": {
            "type": "array"
        },
        "two": {
```

```
            "type": "string"
        }
    },

    # this will match any keys that were not listed in 'properties'
    "additionalProperties": {
        "type": "null"
    }
}
validictory.validate(data, schema)
```

items Provide a schema or list of schemas to match against a list.

If the provided value is a schema object then every item in the list will be validated against the given schema.

If the provided value is a list of schemas then each item in the list must match the schema in the same position of the list. (extra items will be validated according to `additionalItems`)

```
# given a schema object, every list will be validated against it.
data = json.loads(''' {"results": [1, 2, 3, 4, 5]}''')

schema =    {
                "properties": {
                    "results": {
                        "items": {
                            "type": "integer"
                        }
                    }
                }
            }
validictory.validate(data, schema)

# given a list, each item in the list is matched against the schema
# at the same index. (entry 0 in the json will be matched against entry 0
# in the schema, etc)
dataTwo = json.loads(''' {"results": [1, "a", false, null, 5.3]}  ''')
schemaTwo = {
                "properties": {
                    "results": {
                        "items": [
                            {"type": "integer"},
                            {"type": "string"},
                            {"type": "boolean"},
                            {"type": "null"},
                            {"type": "number"}
                        ]
                    }
                }
            }
validictory.validate(dataTwo, schemaTwo)
```

additionalItems Used in conjunction with `items`. If False then no additional items are allowed, if a schema is provided then all additional items must match the provided schema.

```
data = json.loads(''' {"results": [1, "a", false, null, null, null]}  ''')
schema = {
                "properties": {
                    "results": {
                        "items": [
```

```
                                {"type": "integer"},
                                {"type": "string"},
                                {"type": "boolean"}
                            ],

                            # when using 'items' and providing a list (so that values in the list get va
                            # by the schema at the same index), any extra values get validated using addi
                            "additionalItems": {
                                "type": "null"
                            }
                        }
                    }
                }
validictory.validate(data, schema)
```

required If True, the property must be present to validate.

The default value of this parameter is set on the call to *validate()*. By default it is `True`.

```
data = json.loads(''' {"one": 1, "two": 2}''')

schema = {
    "type": "object",
    "properties": {
        "one": {
            "type": "number",
        },
        "two": {
            "type": "number",
        },
        # even though "three" is missing, it will pass validation
        # because required = False
        "three": {
            "type": "number",
            "required": False
        }
    }
}
validictory.validate(data, schema)
```

Note: If you are following the JSON Schema spec, this diverges from the official spec as of v3. If you want to validate against v3 more correctly, be sure to set `required_by_default` to False.

dependencies Can be a single string or list of strings representing properties that must exist if the given property exists.

For example:

```
schema = {"prop01": {"required":False},
          "prop02": {"required":False, "dependencies":"prop01"}}

# would validate
{"prop01": 7}

# would fail (missing prop01)
{"prop02": 7}
```

minimum and maximum If the value is a number (int or float), these methods will validate that the values are less

than or greater than the given minimum/maximum.

Minimum and maximum values are inclusive by default.

```
data = json.loads(''' {"result": 10, "resultTwo": 12}''')

schema = {
    "properties": {
        "result": { # passes
            "minimum": 9,
            "maximum": 10
        },
        "resultTwo": { # fails
            "minimum": 13
        }
    }
}
```

exclusiveMinimum and exclusiveMaximum If these values are present and set to True, they will modify the `minimum` and `maximum` tests to be exclusive.

```
data = json.loads(''' {"result": 10, "resultTwo": 12, "resultThree": 15}''')

schema = {
    "properties": {
        "result": { # fails, has to > 10
            "exclusiveMaximum": 10
        },
        "resultTwo": { # fails, has to be > 12
            "exclusiveMinimum": 12
        },
        "resultThree": { # passes
            "exclusiveMaximum": 20,
            "exclusiveMinimum": 14
        }
    }
}
```

minItems, minLength, maxItems, and maxLength If the value is a list or str, these will test the length of the list or string.

There is no difference in implementation between the items/length variants.

```
data = json.loads(''' { "one": "12345", "two": "2345", "three": [1, 2, 3, 4, 5]} ''')

schema = {

    "properties": {

        "one": { # passes
            "minLength": 4,
            "maxLength": 6
        },

        "two": { # fails
            "minLength": 6
        },
        "three": { # passes
            "maxItems": 5
        }
```

```
    }
}
```

uniqueItems Indicate that all attributes in a list must be unique.

```
data = json.loads(''' {"one": [1, 2, 3, 4], "two": [1, 1, 2]} ''')

schema = {
    "properties": {
        "one": { # passes
            "uniqueItems": True
        },
        "two": { # fails
            "uniqueItems": True
        }
    }
}
```

pattern If the value is a string, this provides a regular expression that the string must match to be valid.

```
data = json.loads(''' {"twentyOne": "21", "thirtyThree": "33"} ''')

schema = {
    "properties": {
        "thirtyThree": {
            "pattern": "^33$"
        }
    }
}
```

blank If False, validate that string values are not blank (the empty string).

The default value of this parameter is set when initializing *SchemaValidator*. By default it is `False`.

```
data = json.loads(''' {"hello": "", "testing": ""}''')

schema = {
    "properties": {
        "hello": {
            "blank": True # passes
        },
        "testing": {
            "blank": False # fails
        }
    }
}
```

enum Provides an array that the value must match if present.

```
data = json.loads(''' {"today": "monday", "tomorrow": "something"}''')

dayList = ["monday", "tuesday", "wednesday", "thursday", "friday", "saturday", "sunday"]
schema = {
    "properties": {
        "today": {
            "enum": dayList # passes
        },
        "tomorrow": {
            "enum": dayList # does not pass, 'something' is not in the enum.
        }
```

```
    }
}
```

format Validate that the value matches a predefined format.

By default several formats are recognized:

- `date-time`: 'yyyy-mm-ddhh:mm:ssZ'
- `date`: 'yyyy-mm-dd'
- `time`: 'hh:mm::ss'
- `utc-millisec`: number of seconds since UTC
- `ip-address`: IPv4 address, in dotted-quad string format (for example, '123.45.67.89')

formats can be provided as a dictionary (of type {"formatString": format_func}) to the `format_validators` argument of `validictory.validate`.

Custom formatting functions have the function signature `format_func(validator, fieldname, value, format_option):`.

- `validator` is a reference to the SchemaValidator (or custom validator class if you passed one in for the `validator_cls` argument in `validictory.validate`).
- `fieldname` is the name of the field whose value you are validating in the JSON.
- `value` is the actual value that you are validating
- `format_option` is the name of the format string that was provided in the JSON, useful if you have one format function for multiple format strings.

Here is an example of writing a custom format function to validate UUIDs:

```
import json
import validictory
import uuid

data = json.loads(''' { "uuidInt": 117574695023396164616661330147169357159,
                        "uuidHex": "fad9d8cc11d64578bff327df93276964"}''')

schema = {
    "title": "My test schema",
    "properties": {
        "uuidHex": {
            "format": "uuid_hex"
        },
        "uuidInt": {
            "format": "uuid_int"
        }
    }
}

def validate_uuid(validator, fieldname, value, format_option):

    print("**********************")
    print("validator:",validator)
    print("fieldname:", fieldname)
    print("value", value)
    print("format_option", format_option)
    print("**********************")
```

```
    if format_option == "uuid_hex":
        try:
            uuid.UUID(hex=value)
        except Exception as e:
            raise validictory.FieldValidationError("Could not parse UUID \
            from hex string %(uuidstr)s, reason: %(reason)s"
                % {"uuidstr": value, "reason": e}, fieldname, value)

    elif format_option == "uuid_int":
        try:
            uuid.UUID(int=value)
        except Exception as e:
            raise validictory.FieldValidationError("Could not parse UUID \
            from int string %(uuidstr)s, reason: %(reason)s"
                % {"uuidstr": value, "reason": e}, fieldname, value)
    else:
        raise validictory.FieldValidationError("Invalid format option for \
        'validate_uuid': %(format)s" % format_option,
            fieldName, value)

try:
    formatdict = {"uuid_hex": validate_uuid, "uuid_int": validate_uuid}
    validictory.validate(data, schema, format_validators=formatdict)
    print("Successfully validated %(data)s!" % {"data": data})
except Exception as e2:
    print("couldn't validate =( reason: %(reason)s" % {"reason": e})
```

divisibleBy Ensures that the data value can be divided (without remainder) by a given divisor (**not 0**).

```
data = json.loads('''{"value": 12, "valueTwo": 13} ''')

schema = {
    "properties": {
        "value": {
            "divisibleBy": 2 # passes
        },
        "valueTwo": {
            "divisibleBy": 2 # fails
        }
    }
}
```

title and description These do no validation, but if provided must be strings or a `~validictory.SchemaError` will be raised.

```
data = json.loads(''' {"hello": "testing"}''')

schema = {
    "title": "My test schema",
    "properties": {
        "hello": {
            "type": "string",
            "description": Make sure the 'hello' key is a string"
        }
    }
}
```

3.1.3 Examples

Using a Schema

The schema can be either a deserialized JSON document or a literal python object

```
data = json.loads(''' {"age": 23, "name": "Steven"} ''')

# json string
schemaOne = json.loads(''' {"type": "object", "properties":
    {"age": {"type": "integer"}, "name": {"type": "string"}}} ''')

# python object literal
schemaTwo = {"type": "object", "properties":
    {"age": {"type": "integer"}, "name": {"type": "string"}}}

validictory.validate(data, schemaOne)
validictory.validate(data, schemaTwo)
```

Validating Using Builtin Types

```
data = json.loads('''

    {
        "name": "bob",
        "age": 23,
        "siblings": null,
        "registeredToVote": false,
        "friends": ["Jane", "Michael"],
        "heightInInches": 70.2
    }   ''')

schema =
    {
        "type": "object",
        "properties": {
            "name": {
                "type": "string"
            },
            "age": {
                "type": "integer"
            },
            "siblings": {
                "type": "null"
            },
            "registeredToVote": {
                "type": "boolean"
            },
            "friends": {
                "type": "array"
            }
        }
    }

validictory.validate(data, schema)
```

the 'number' type can be used when you don't care what type the number is, or 'integer' if you want a non floating point number

```
dataTwo = json.loads('''{"valueOne": 12} ''')

schemaTwo = { "properties": {  "valueOne": { "type": "integer"}} }

validictory.validate(dataTwo, schemaTwo)
```

the 'any' type can be used to validate any type.

```
dataThree = json.loads(''' {"valueOne": 12, "valueTwo": null, "valueThree": "hello" }''')

schemaThree = {
    "properties": {
        "valueOne": {"type": "any"},
        "valueTwo": {"type": "any"},
        "valueThree": {"type": "any"}
    }
}

validictory.validate(dataThree, schemaThree)
```

You can list multiple types as well.

```
dataFour = json.loads(''' {"valueOne": 12, "valueTwo": null}''')

schemaFour =  {
    "properties": {
        "valueOne": {
            "type": ["string", "number"]
        },
        "valueTwo": {
            "type": ["null", "string"]
        }
    }
}

validictory.validate(dataFour, schemaFour)
```

Validating Nested Containers

```
data = json.loads('''
    {
        "results": {
            "xAxis": [
                [0, 1],
                [1, 3],
                [2, 5],
                [3, 1]
            ],
            "yAxis": [
                [0, "sunday"],
                [1, "monday"],
                [2, "tuesday"],
                [3, "wednesday"]
            ]
        }
```

```
    } ''')

schema = {

    "type": "object",
    "properties": {
        "results": {

            "type": "object",
            "properties": {
                "xAxis": {
                    "type": "array",
                    "items": {
                        "type": "array",
                        # use a list of schemas, so that the the schema at index 0
                        # matches the item in the list at index 0, etc.
                        "items": [{"type": "number"}, {"type": "number"}]
                    }
                },
                "yAxis": {
                    "type": "array",
                    "items": {
                        "type": "array",
                        "items": [{"type": "number"}, {"type": "string"}]
                    }
                }
            }
        }
    }
}
validictory.validate(data, schema)
```

Specifying Custom Types

If a list is specified for the 'types' option, then you can specify a schema or multiple schemas that each element in the list will be tested against. This also allows you to split up your schema definition for ease of reading, or to share schema definitions between other schemas.

```
schema = {
    "type": "object",
    "properties": {
        "foo_or_bar_list": {
            "type": "array",
            "items": {
                "type": [
                    {"type": "object",
                     # foo definition
                    },
                    {"type": "object",
                      # bar definition
                    },
                ]
            }
        }
    }
}
```

A common example of this is the GeoJSON spec, which allows for a geometry collection to have a list of geometries (Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon).

Simplified GeoJSON example:

```
# to simplify things we make a few subschema dicts

position = {
    "type": "array",
    "minItems": 2,
    "maxItems": 3
}

point = {
    "type": "object",
    "properties": {
        "type": {
            "pattern": "Point"
        },
        "coordinates": {
            "type": position
        }
    }
}

multipoint = {
    "type": "object",
    "properties": {
        "type": {
            "pattern": "MultiPoint"
        },
        "coordinates": {
            "type": "array",
            "minItems": 2,
            "items": position
        }
    }
}

# the main schema
simplified_geojson_geometry = {
    "type": "object",
    "properties": {
        "type": {
            "pattern": "GeometryCollection"
        },
        # this defines an array ('geometries') that is a list of objects
        # which conform to one of the schemas in the type list
        "geometries": {
            "type": "array",
            "items": {"type": [point, multipoint]}
        }
    }
}
```

(thanks to Jason Sanford for bringing this need to my attention, see his blog post on validating GeoJSON)

3.2 validictory module

3.2.1 validate

validictory.**validate**(*data*, *schema*, *validator_cls=<class 'validictory.validator.SchemaValidator'>*, *format_validators=None*, *required_by_default=True*, *blank_by_default=False*, *disallow_unknown_properties=False*, *apply_default_to_data=False*, *fail_fast=True*, *remove_unknown_properties=False*)

Validates a parsed json document against the provided schema. If an error is found a `ValidationError` is raised.

If there is an issue in the schema a `SchemaError` will be raised.

Parameters

- **data** – python data to validate
- **schema** – python dictionary representing the schema (see **'schema format'**_)
- **validator_cls** – optional validator class (default is `SchemaValidator`)
- **format_validators** – optional dictionary of custom format validators
- **required_by_default** – defaults to True, set to False to make `required` schema attribute False by default.
- **disallow_unknown_properties** – defaults to False, set to True to disallow properties not listed in the schema definition
- **apply_default_to_data** – defaults to False, set to True to modify the data in case the schema definition includes a "default" property
- **fail_fast** – defaults to True, set to False if you prefer to get all validation errors back instead of only the first one
- **remove_unknown_properties** – defaults to False, set to True to filter out properties not listed in the schema definition. Only applies when disallow_unknown_properties is False.

3.2.2 SchemaValidator

class validictory.**SchemaValidator**(*format_validators=None*, *required_by_default=True*, *blank_by_default=False*, *disallow_unknown_properties=False*, *apply_default_to_data=False*, *fail_fast=True*, *remove_unknown_properties=False*)

Validator largely based upon the JSON Schema proposal but useful for validating arbitrary python data structures.

Parameters

- **format_validators** – optional dictionary of custom format validators
- **required_by_default** – defaults to True, set to False to make `required` schema attribute False by default.
- **blank_by_default** – defaults to False, set to True to make `blank` schema attribute True by default.
- **disallow_unknown_properties** – defaults to False, set to True to disallow properties not listed in the schema definition

- **apply_default_to_data** – defaults to False, set to True to modify the data in case the schema definition includes a "default" property
- **fail_fast** – defaults to True, set to False if you prefer to get all validation errors back instead of only the first one
- **remove_unknown_properties** – defaults to False, set to True to filter out properties not listed in the schema definition. Only applies when disallow_unknown_properties is False.

3.2.3 Exceptions

class `validictory.`**`ValidationError`**
validation errors encountered during validation (subclass of `ValueError`)

class `validictory.`**`SchemaError`**
errors encountered in processing a schema (subclass of `ValueError`)

3.3 validictory changelog

3.3.1 1.0.1

2015-09-24

- bugfix for fail_fast w/ type lists

3.3.2 1.0.0

2015-01-16

- stable release

3.3.3 1.0.0a2

2014-07-15

- ensure path to field is used in error

3.3.4 1.0.0a1

2014-07-10

- fix TypeError from format validators
- some documentation fixes
- enum options are callable (from James McKinney)
- switch to py.test
- internal changes to how _validate and _error work
- initial work on fail_fast=False
- initial work on descriptive field names

3.3.5 0.9.3

2013-11-25

- fix bad 0.9.2 release that didn't have a fix for invalid code from a PR

3.3.6 0.9.2

2013-11-25

- fix from Marc Abramowitz for validating dict-like things as dicts
- fix for patternProperties from Juan Menéndez & James Clemence
- include implementation of "default" property from Daniel Rech
- drop official support for Python 3.2
- remove a test that relied on dict ordering
- updated docs from Mark Grandi
- fix where format validators were cleared (also Mark Grandi)

3.3.7 0.9.1

2013-05-23

- fix for error message when data doesn't match one of multiple subtypes
- fix for disallow_unknown_properties

3.3.8 0.9.0

2013-01-19

- remove optional and requires, deprecated in 0.6
- improved additionalProperties error message
- improved schema error message
- add long to utc-millisec validation
- accept Decimal where float is accepted
- add FieldValidationError so that field names can be retrieved from error
- a few Python 3 fixes

3.3.9 0.8.3

2012-03-13

- bugfix for Python 3: fix regression from 0.8.1 in use of long

3.3.10 0.8.2

2012-03-09

- doc improvements
- PEP8 nearly everything
- bugfix for patternProperties
- ip-address should have been a format, not a type, breaks any code written depending on it in 0.8.1

3.3.11 0.8.1

2012-03-04

- add GeoJSON example to docs
- allow longs in int/number validation
- ignore additionalProperties for non-dicts
- ip-address type validator

3.3.12 0.8.0

2012-01-26

- validate_enum accepts any container type
- add support for Python 3
- drop support for Python 2.5 and earlier

3.3.13 0.7.2

2011-09-27

- add blank_by_default argument
- more descriptive error message for list items

3.3.14 0.7.1

2011-05-03

- PEP8 changes to code base
- fix for combination of format & required=False
- use ABCs to determine types in Python >= 2.6

3.3.15 0.7.0

2011-03-15

- fix dependencies not really supporting lists
- add what might be the draft03 behavior for schema dependencies
- add Sphinx documentation

3.3.16 0.6.1

2011-01-21

- bugfix for uniqueItems

3.3.17 0.6.0

2011-01-20

- more draft-03 stuff: patternProperties, additionalItems, exclusive{Minimum,Maximum}, divisibleBy
- custom format validators
- treat tuples as lists
- replace requires with dependencies (deprecating requires)
- replace optional with required (deprecating optional)
- addition of required_by_default parameter

3.3.18 0.5.0

2011-01-13

- blank false by default
- draft-03 stuff: uniqueItems, date formats

3.3.19 0.4.1

2010-08-27

- test custom types
- optional defaults to False correctly
- remove raise_errors
- add value check in additionalProperties

3.3.20 0.4.0

2010-08-02

- renamed to validictory
- removal of maxDecimal
- ignore unknown attributes
- differentiate between a schema error and a validation error
- filter through _error
- combine Items/Length checks
- modular type checking
- major test refactor

3.3.21 0.3.0

2010-07-29

- took over abandoned json_schema code
- removal of interactive mode
- PEP 8 cleanup of source
- list/dict checks more flexible
- remove identity/options/readonly junk

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

V

S

V