dschema Documentation

Release 0.3.1.3a1

Teriks

Sep 14, 2019

API Documentation

CHAPTER 1

dschema package

1.1 Module Contents

class dschema.**Namespace**(*dictionary*)
Bases: object

Simple dynamic object, optimized for dschema.

exception dschema.**ValidationError**(*message*)
Bases: Exception

Thrown on any validation error, such as missing required properties.

exception dschema.**MissingKeyError**(*message*, *namespace*)
Bases: dschema.dschema.ValidationError

Thrown when a required namespace/property defined in the schema is missing from data being validated.

namespace
Full path to the missing key.

exception dschema.**ExtraKeysError**(*message*, *keys*)
Bases: dschema.dschema.ValidationError

Thrown when extra keys exist in data being validated which do not exist in the schema.

keys
A set containing all the extraneous keys.

exception dschema.**TypeValidationError**(*message*, *type_exception=None*)
Bases: dschema.dschema.ValidationError

Thrown when a type validation function throws on an incoming value.

type_exception
The exception object raised from your validation function.

exception dschema.**SchemaError**(*message*)
Bases: Exception

Thrown for errors related to the schema definition itself.

exception dschema.**SchemaDefaultError**(*validation_error*)
Bases: dschema.dschema.SchemaError

Thrown when a default value for a property/namespace is invalid for its own schema.

validation_error
The *ValidationError* instance raised when validating the default value against the schema failed.

exception dschema.**SchemaMissingTypeError**(*message*, *typename*)
Bases: dschema.dschema.SchemaError

Thrown when an undefined type is referenced by name in a schema.

typename
The referenced type name.

dschema.**prop**(***kwargs*)
Helper that returns a schema property specification from keyword arguments.

Parameters ****kwargs** – See below

Keyword Arguments

- *default* (object) – Default value when none is present in the validated dictionary.
- *required* (object) – Is this key/value pair required to be present?
- *type* (callable) – The validation callable to use on the incoming value.
- *dict* (bool) – Should the value be interpreted as a raw nested dictionary?

class dschema.**Validator**(*schema*)
Bases: object

Schema validator class

add_type(*name*, *validation_function*)
Register a type validation callable that can be referenced by name in the schema.

See: *types*

Parameters

- **name** – Name which can be referenced in the schema using a string.
- **validation_function** – Associated validation function.

remove_type(*name*)
Remove an existing type validation callable.

See: *add_type()* and *types*

Parameters **name** – The name previously registered with *add_type()*

schema
schema dictionary (reassignable).

types
Types dictionary, should contain type validator callables by name (reassignable).

Must always implement `get(key, default)` and `types['key']` like a python `dict`.

validate(*dictionary*, *copy=True*, *namespace=False*, *extra_keys=False*)
Validate a dictionary object using the defined schema and return it a copy.

Defaults defined in the schema will be filled out if they do not exist in the incoming data.

Parameters

- **copy** – Whether or not to deep copy the input dictionary before processing, if this is not done, then the input dictionary will be modified to a useless state. validate can run faster if do not plan to use the input dictionary again and you use **copy=False**.
- **dictionary** – (`dict`) object to validate.
- **namespace** – If `True`, return a deserialized *dschema.Namespace* object.
- **extra_keys** – Allow extra key value pairs that do not exist in the schema to pass through without exception. In effect, only run validation on keys which are found to exist in the schema, and let others always pass through if they have no schema defined for them.

Returns Processed input dictionary

CHAPTER 2

About

dschema is a small library for validating the content of python dictionary objects against a schema.

The schema can be defined in code or entirely as text (parsed from JSON generally)

dschema was made for validating config files written in JSON, and allows for specifying required and default property values, and also custom type validation.

CHAPTER 3

Usage

3.1 Defining schema with code

```
import re
import phonenumbers
import dschema


# https://github.com/daviddrysdale/python-phonenumbers
# pip install phonenumbers


def phone_type(number):
    # Exceptions are validation errors
    # Very similar design to the "argparse" module
    return phonenumbers.parse(number)


def ssn_type(ssn):
    if re.match('^\d{3}-?\d{2}-?\d{4}$', ssn):
        return ssn
    else:
        raise ValueError('"{}" is not a valid SSN.')


schema = {
    'person': {
        'first_name': dschema.prop(required=True),
        'last_name': dschema.prop(required=True),
        'phone': dschema.prop(required=True, type=phone_type),
        'ssn': dschema.prop(required=True, type='ssn_type'),

        dschema.Required: True
        # "person" namespace is required, you must specify
```

(continues on next page)

(continued from previous page)

```
        # even if "person" itself contains required properties
    },

    # Allow a raw dictionary value to pass through

    'other_info': dschema.prop(default=dict(), dict=True),

    # default to False if not present

    'subscribed': dschema.prop(default=False, type=bool)
}

validator = dschema.Validator(schema)

# you can use this to add types that are recognized by name.
# which is useful if you want your schema to be entirely textual

validator.add_type('ssn_type', ssn_type)

# you will need to define default types on your own
# if you want to reference them by name

# validator.add_type('int', int)


data = {
    'person': {
        'first_name': "John",
        'last_name': "Smith",
        'phone': '+1 234 5678 9000',
        'ssn': '123-45-6789'
    },

    'other_info': {
        'website': 'www.johnsmith.com',
    }
}

# If return_namespace is left False, a plain dictionary is returned

result = validator.validate(data, namespace=True)

print(result)

# Prints: (un-indented)

# Namespace(
#     person=Namespace(
#         first_name='John',
#         last_name='Smith',
#         phone=PhoneNumber(...),
#         ssn='123-45-6789'),
#     other_info={'website': 'www.johnsmith.com'},
#     subscribed=False
# )
```

(continues on next page)

(continued from previous page)

```
# Each Namespace is just a dynamic object

print(result.person.first_name)  # -> John
print(result.person.last_name)  # -> Smith

print(result.person.phone)
# - > Country Code: 1 National Number: 23456789000

print(result.person.ssn)  # -> 123-45-6789

print(result.other_info)  # -> {'website': 'www.johnsmith.com'}

print(result.subscribed)  # -> False (default)
```

3.2 Defining schema with text

```
import json
import re
import phonenumbers
import dschema


def phone_type(number):
    return phonenumbers.parse(number)


def ssn_type(ssn):
    if re.match('^\d{3}-?\d{2}-?\d{4}$', ssn):
        return ssn
    else:
        raise ValueError("'{}' is not a valid SSN.")


schema = """{
    "person": {
        "first_name": {"@required": true},
        "last_name": {"@required": true},
        "phone": {"@required": true, "@type": "phone_type"},
        "ssn": {"@required": true, "@type": "ssn_type"},
        "@required": true
    },

    "other_info": {"@dict": true, "@default": {} },

    "subscribed": {"@default": false}
}"""

# Load schema from json this time
validator = dschema.Validator(json.loads(schema))

validator.add_type('phone_type', phone_type)
validator.add_type('ssn_type', ssn_type)

data = {
```

(continues on next page)

(continued from previous page)

```
    'person': {
        'first_name': 'John',
        'last_name': 'Smith',
        'phone': '+1 234 5678 9000',
        'ssn': '123-45-6789'
    },

    'other_info': {
        'website': 'www.johnsmith.com',
    }
}

result = validator.validate(data, namespace=True)

print(result)

# Prints: (un-indented)

# Namespace(
#     person=Namespace(
#         first_name='John',
#         last_name='Smith',
#         phone=PhoneNumber(...),
#         ssn='123-45-6789'),
#     other_info={'website': 'www.johnsmith.com'},
#     subscribed=False
# )

print(result.person.first_name)  # -> John
print(result.person.last_name)  # -> Smith

print(result.person.phone)
# - > Country Code: 1 National Number: 23456789000

print(result.person.ssn)  # -> 123-45-6789

print(result.other_info)  # -> {'website': 'www.johnsmith.com'}

print(result.subscribed)  # -> False (default)
```

3.3 Defining default values

```
import dschema

# specifying defaults in a nested property
# will completely fill the property tree
# if it does not exist in your data

schema = {
    'a': {
        'b': {
            'c': dschema.prop(default='d')
        }
    }
```

(continues on next page)

(continued from previous page)

```
}

r = dschema.Validator(schema).validate({}, namespace=True)

print(r.a.b.c)  # -> prints: d

# you can define a default value for nodes with
# nested properties, the default value must match
# the schema of the node it is defined in

schema = {
    'a': {
        dschema.Default: {'b': {'c': 1}},
        'b': {
            'c': dschema.prop(type=int)
        }
    }
}

r = dschema.Validator(schema).validate({}, namespace=True)

print(r.a.b.c)  # -> prints: 1

# this is an error because the default value
# does not match the schema

schema = {
    'a': {
        dschema.Default: {'b': {'c': 'notint'}},
        'b': {
            'c': dschema.prop(type=int)
        }
    }
}

try:
    # exception here..
    dschema.Validator(schema).validate({'a': None})
except dschema.SchemaDefaultError as e:
    print(e)

# you validator functions return values are
# always considered when processing default values

schema = {
    'a': {
        dschema.Default: {'b': {'c': 1}},
        'b': {
            'c': dschema.prop(type=lambda x: x + 1)
        }
    }
}

r = dschema.Validator(schema).validate({}, namespace=True)

# ===
```

(continues on next page)

(continued from previous page)

```
print(r.a.b.c)  # prints: 2

schema = {
    'a': dschema.prop(default=1, type=lambda x: x + 1)
}

r = dschema.Validator(schema).validate({'a': None}, namespace=True)

print(r.a)  # prints: 2
```

3.4 Handling validation errors

```
import dschema

validator = dschema.Validator({
    'app_auth': {
        'id': dschema.prop(required=True),
        'token': dschema.prop(required=True),

        dschema.Required: True
        # You must specify a namespace as required
        # if you want it to be required, even if it
        # contains required properties
    },

    'integer': dschema.prop(required=True, type=int)
})


# Handling of missing required values
# ===================================

data = {
    'app_auth': {
        'token': 'somerandomthing',
    },
    'integer': 1
}

try:
    validator.validate(data)
except dschema.MissingKeyError as e:
    # message about 'app_auth.id' being required but missing...
    print(e)

data = {'integer': 1}

try:
    validator.validate(data)
except dschema.MissingKeyError as e:
    # message about 'app_auth' being required but missing...
    print(e)
```

(continues on next page)

(continued from previous page)

```
# Handling type validation errors
# ================================

data = {
    'app_auth': {
        'id': 12345,
        'token': 'somerandomthing'
    },
    'integer': 'notinteger'
}

try:
    validator.validate(data)
except dschema.ValidationError as e:
    # message about 'integer' failing type validation...
    print(e)

try:
    validator.validate(data)
except dschema.TypeValidationError as e:
    # be more specific and cache the TypeValidationError

    # message about 'integer' failing type validation...
    print(e)

    # print the exception that came out of the validator function
    print(e.type_exception)


# Handling of extraneous key values
# ==================================

data = {
    'app_auth': {
        'id': 12345,
        'token': 'somerandomthing',
        'extra_stuff': 'should not be here'
    },
    'integer': 1
}


try:
    validator.validate(data)
except dschema.ExtraKeysError as e:
    # Message about:
    # namespace 'app_auth' containing extraneous keys {'extra_stuff'}
    print(e)


# Allow extra keys to pass through into the result...
result = validator.validate(data, extra_keys=True)

print(result['app_auth']['extra_stuff'])  # -> prints: should not be here
```

3.5 Handling schema definition errors

```
import dschema


# specifying a required property as
# having a default value.

schema = {
    'bad': dschema.prop(required=True, default='cant-have-both')
}

try:
    dschema.Validator(schema).validate({'bad': 'stuff'})
except dschema.SchemaError as e:

    # Message about 'required' and 'default' being mutually exclusive
    print(e)


# Not providing a validation handler
# for a type specified as a string

schema = {
    'no_type_validator': dschema.prop(required=True, type='int')
}

try:
    validator = dschema.Validator(schema)

    # Validator.add_type must be used to add
    # something that handles 'int' ...

    # validator.add_type('int', int)

    validator.validate({'no_type_validator': 1})

except dschema.SchemaMissingTypeError as e:

    # Message about:
    # 'no_type_validator' schema type callable 'int' not provided.
    print(e)

# providing a default value that does
# not validate against the schema

schema = {
    'node':
        {
            dschema.Default: {'bad': {'prop': 'notint'}},
            'bad': {
                'prop': dschema.prop(type=int)
             }
        }
}

try:
```

(continues on next page)

(continued from previous page)

```
    validator = dschema.Validator(schema)

    validator.validate({})

except dschema.SchemaDefaultError as e:

    # Message about:
    # 'bad.prop' failed type validation: invalid literal for
    # int() with base 10: 'notint'
    print(e)
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index