JSON Document Documentation

Release 0.7

Zygmunt Krynicki

February 16, 2012

CONTENTS

This package provides intuitive and powerful binding system for JSON documents. It bridged the gap between raw python objects, json schema and json files. A powerful default system also allows developers to access an empty document and see the default values from the schema without any code changes.

CHAPTER

ONE

INSTALLATION

You can install `json-document` from source using pip. Please note that currently the code is in **alpha** stages and is not recommended outside of the early adopter group.

CHAPTER

TWO

TABLE OF CONTENTS

2.1 Basic features

If you have a few seconds, just read the next two sub-sections.

2.1.1 Setting and accessing value

You can use a Document class much like a normal python object. If you are using JSON via raw dictionaries/lists you'll find that a lot of things are the same:

```
>>> from json_document.document import Document
>>> doc = Document({})
>>> doc['hello'] = 'world'
```

The first major difference is evident when you want to refer to data. Instead of returning the value directly you get an instance of `DocumentFragment`. To access the value you'll have to use the `value` property:

```
>>> doc['hello'].value
'world'
>>> doc.value
{'hello': 'world'}
```

Accessing the fragment directly is also possible and indeed is where a lot of the value json_document brings is hidden. To learn about that though you'll have to learn about using schema with your documents.

2.1.2 Using document schema

Schema defines how a valid document looks like. It is constructed of a series of descriptions (written in JSON itself) and is quite powerful in what can be expressed. If you have no experience with JSON-Schema you can think of it as DTD for XML on steroids. Don't worry it's easy to learn by example.

Let's design a simple schema for a personnel registry. Each document will describe one person and will remember their name and age:

```
>>> person_schema = {
...     "type": "object",
...     "title": "Person record",
...     "description": "Simplified description of a person",
...     "properties": {
...         "name": {
...             "type": "string",
```

```
...             "title": "Full name"
...         },
...         "age": {
...             "type": "number",
...             "title": "Age in years"
...         }
...     }
... }
```

This schema can be read as follows:

- The root element is an object titled "Person record".
- It has a property "name" that is a string titled "Full name".
- It also has a property "age" that is a number titled "Age in years"

This schema is very simple but it already defines the correct shape of a document. It defines the type of the root object (a json "object", for python that translates to a dictionary instance). It also describes the attributes of that object (name and age) and their type. The schema also mixes documentation elements via the "title" and "description" properties.

Using a schema you can *validate* documents for correctness. Let's see how that works:

```
>>> joe = Document({"name": "joe", "age": 32}, person_schema)
>>> joe.validate()
```

Calling `validate()` would have raised an exception if joe was not a valid "Person record". Let's set the age to an invalid type to see how that works:

```
>>> joe["age"] = "thirty two"
>>> joe.validate()
Traceback (most recent call last):
...
ValidationError: ValidationError: Object has incorrect type (expected number) object_expr='object.age
```

Boom! Not only did the validation fail. We've got a detailed error message that outlines the problem. It also gives us the JavaScript expression that describes the part that did not match the schema (`object.age`) and the part of the schema that was violated (`schema.properties.age.type`).

Because the actual value is hidden behind the .value property we can stash a set of useful properties and methods in each `DocumentFragment`. One of them is `.schema` which unsurprisingly returns the associated schema element (if we have one). Instead of returning the raw JSON schema it returns a smart wrapper around it that has properties corresponding to each legal schema part (such as `.type` and `.properties`). You can use it to access meta-data such as title and description:

```
>>> joe["age"].schema.title
'Age in years'
>>> joe.schema.description
'Simplified description of a person'
```

You can also access things like type but be aware that it has some quirks. Refer to json-schema-validator documentation for details on the Schema class. For example, the type is automatically converted to a list of valid types:

```
>>> joe["name"].schema.type
['string']
```

One useful property is .schema.optional which tells if if an element is required or not. By default everything is required, unless marked optional:

```
>>> joe["name"].schema.optional
False
```

2.2 Core features

So now you know roughly about documents and schema. You know that accessing items on a document instance returns `DocumentFragment` objects (that have a `value` and `schema` properties) but setting items sets the value directly. You know that a document may have an associated schema and that calling `validate()` checks for errors.

2.2.1 Supported types

Now let's expand that. So far we've only used objects (that map to Python dictionaries). We can use the following types in our documents:

- Dictionaries (JSON objects, schema type "object")
- Lists (JSON arrays, schema type "array")
- Strings (and Unicode strings, schema type "string")
- Integers, floating point numbers and Decimals (JSON numbers, schema types "integer", "number")
- True and False (JSON true and false values, schema type "boolean")
- None (JSON null value, schema type "null")

You can use any of those items as the root object:

```
>>> from json_document.document import Document

>>> shopping_list = Document([])
>>> shopping_list.value.append("milk")
>>> shopping_list.value.append("cookies")
>>> shopping_list.value
['milk', 'cookies']

>>> yummy = Document("json")
>>> yummy.value
'json'

>>> life = Document(42)
>>> life.value
42

>>> long_example = Document(True)
>>> long_example.value
True

>>> surprise = Document(None)
>>> surprise.value
None
```

2.2.2 Default schema

All documents have a schema, even if you don't specify one. By default the schema describes an arbitrary JSON value (one of any type):

```
>>> doc = Document({})
>>> doc.schema
Schema({'type': 'any'})
```

This does not apply to fragments you create yourself. Those always inherit the schema from their parent document (depending on the item used to create or access that fragment). Since the default schema does not describe the `foo` property it is assigned an empty schema instead:

```
>>> doc['foo'] = 'bar'
>>> doc['foo'].schema
Schema({})
```

It's important to point out that default type is `any`. It allows the value to be of any previously mentioned type:

```
>>> doc['foo'].schema.type
['any']
```

2.2.3 Schema on fragments

It's pretty obvious but important to point out that when a schema describes a document and you access a fragment of that document the fragment's schema is the corresponding fragment of the whole:

```
>>> doc = Document({"foo": "bar"}, {"properties": {"foo": {"type": "string"}}})
>>> doc["foo"].schema
Schema({'type': 'string'})
```

This is very useful when you consider that a schema can specify default values for missing elements.

2.2.4 Using default values

Having a schema for a document is not only useful because you can validate it. It is also useful because you can embed default values in the schema and transparently use them as if they were specified in the document.

Let's see how this works. Imagine a simple application that has a *save on exit* feature. The application starts up, loads settings from a configuration file and does something useful. When the user quits the application it can save the current document without asking for confirmation. Traditionally you'd embed the default value in the code of your application. If you were smart you'd build an API for your configuration to transparently provide the default for you (or you'd generate the default configuration file if it was missing).

Both of those approaches are not very nice in practice. The former requires you to build additional layers of API around your basic notion of configuration. The latter prevents you from differentiating default values and settings identical to default values.

We can do better than that. Let's start with describing our configuration schema:

```
>>> schema = {
...     "type": "object",
...     "properties": {
...         "save_on_exit": {
...             "type": "boolean",
...             "default": True,
...             "optional": True
...         }
...     }
... }
```

There are a couple of new elements here:

- The default value is specified, exactly once, in the schema
- The property is marked as optional, when missing the document will still be valid.

Let's create a configuration object to see how this works:

```
>>> config = Document({}, schema)
>>> config["save_on_exit"].value
True
```

Success! Still a little verbose but already doing much, much better. The default value was looked up in the schema and provided in place of our missing configuration option. We can see this option is default by accessing a few methods and properties. With `is_default` you can check if .value is a real thing or a substitute from the schema. With `default_value` you can see what the default is. Lastly, with `default_value_exists` you can check if there even is a default specified. After all, if the schema has no defaults then your code will simply trigger an exception instead:

```
>>> config["save_on_exit"].is_default
True
>>> config["save_on_exit"].default_value
True
>>> config["save_on_exit"].default_value_exists
True
```

We can still change the value as we had before, all of that works as expected. The non-obvious part is what the value of our document is. Before we change anything it is still left as-is, as we provided it initially, that is, empty.:

```
>>> config.value
{}
```

If we change it, however, it reflects that change:

```
>>> config["save_on_exit"] = False
>>> config.value
{'save_on_exit': False}
```

2.2.5 Reverting to defaults

Let's suppose our application wants to provide a "revert to defaults" button that resets all configuration options to what was provided out of the box. JSON document has a sweet feature to support this kind of behavior.

Let's start with some settings we loaded for this user (we are reusing the schema from the previous example):

```
>>> config = Document({"save_on_exit": True}, schema)
```

The first thing to point out is that a default value is a 'special' thing. Being equal to the default value is not the same as being default. Here, the `save_on_exit` option is True, the same as the default from the schema. It is not default though:

```
>>> config["save_on_exit"].is_default
False
```

To really make it default you need to call the `revert_to_default()` method:

```
>>> config["save_on_exit"].revert_to_default()
>>> config["save_on_exit"].value
True
>>> config["save_on_exit"].is_default
True
```

When you do that the document is transformed and the part we've customized is removed. Obviously without a default value in the schema this method would raise an exception with an appropriate message:

```
>>> config.value
{}
```

Defaults are a very powerful system. Used correctly they allow applications to recover from manually edited configuration files (config errors), allow users to customize parts of their configuration while allowing defaults to evolve with future versions and significantly simplify application configuration handling for programmers where less checking is needed, especially when coupled with JSON schema validation that can not only shape but constrain values of specific properties.

2.2.6 Fragments and references

So far in this document we've been referring to document fragments by accessing dictionary items and array elements on the root document object. Accessing those items transparently creates `DocumentFragment` instances. Wrapper objects pointing to a sub-tree of the document object. It is possible to save those references and use them freely for convenience. Let's see how this works:

```
>>> doc = Document({})
>>> doc["list"] = [1, 2, 3]
>>> doc["dict"] = {"hello": "world"}
>>> doc["value"] = "I'm a plain string"
```

For clarity, this is how the document looks like now:

```
>>> doc.value
{'dict': {'hello': 'world'}, 'list': [1, 2, 3], 'value': "I'm a plain string"}
```

Let's obtain a reference to the list:

```
>>> lst = doc["list"]
```

A document fragment is much like a document itself (`Document` is also a DocumentFragment subclass) it has a .value and .schema properties. It has a revert_to_default() method and everything you've learned so far.

It can also be modified, and here it gets interesting. You can modify the value by assigning to the .value property:

```
>>> lst.value
[1, 2, 3]
>>> lst.value = [4, 5]
>>> lst.value
[4, 5]
```

The interesting part is that this automatically integrates into the document this fragment is a part of:

```
>>> doc.value
{'dict': {'hello': 'world'}, 'list': [4, 5], 'value': "I'm a plain string"}
```

In general it you can freely modify the tree and it will work as expected:

```
>>> dct = doc["dict"]
>>> dct.value = {'hello': 'there'}
>>> val = doc["value"]
>>> val.value = 42
>>> doc.value
{'dict': {'hello': 'there'}, 'list': [4, 5], 'value': 42}
```

You can also use mutating methods (those that alter the state of the value), in this case you are not assigning a new value to the .value property but rather calling some method on it:

```
>>> lst.value.append(6)
>>> dct.value['hello'] = 'joe'
>>> doc.value
{'dict': {'hello': 'joe'}, 'list': [4, 5, 6], 'value': 42}
```

Fragments also have a few interesting properties. The .document property allows you to reach the document object this fragment is a part of. The .parent property points to the parent fragment (say, if you have a fragment to member of a list then the .parent will be pointing to the list itself). The .item property is perhaps named confusingly but it is the index of this fragment in the parent fragment (the list index or dictionary key)

Fragments also have few special methods that make using them more natural in python. You can check the length (of strings, dicts and lists), you can check for membership using the `foo in bar` syntax. You can also iterate over containers (lists and dicts only)

2.2.7 Orphaned fragments

Since you can keep references to fragments around for as long as you like it is possible to create an interesting situation. It is only interesting in a problematic way though. A fragment can become orphaned (and useless) when its parent (or its parent, all the way up to the root document object) are overwritten. Let's see how this works:

```
>>> doc = Document({})
>>> doc['foo'] = 'bar'
>>> foo = doc['foo']
>>> doc.value = {}
>>> foo.is_orphaned
True
```

So now the `foo` fragment is an orphaned. A few things happen when this occurs:

- The .document property is set to None
- The .parent property is set to None
- The .value is set to a deep copy of the original value

So for all intents and purposes an orphaned node is independent leftover that is totally disconnected from the original. This means that changing its value is not going to alter the document anymore (since this would make no sense). In fact, attempting to change the value will raise an `OrphanedFragmentError`:

```
>>> foo.value = "barf"
Traceback (most recent call last):
...
OrphanedFragmentError: Attempt to modify orphaned document fragment
```

Usually when you see this it indicates a programming error. If you want to keep using something don't overwrite its parent. For convenience it is not an error to read from an orphaned fragment as it is useful in some cases and provides some level of 'transaction isolation' where you can bet that you've got a working fragment (just that the writes will fail)

2.3 Advanced features

If you got this far you know pretty much everything there is about the basic feature set. The rest of this document will make you more productive by letting you write less code and by letting you write code that is more natural to read and use later.

2.3.1 Custom fragments

Since you get fragment objects every time you access some of its parts would not it be nice to be able to put your custom methods there? This way you could somewhat forget about working with JSON and see this as a part of your class hierarchy.

I thought it was useful so here it is. You need to have a schema for your document the reason for that is we'll be embedding special schema elements that will override the instantiated fragment class. Usually this is simple but it is boiler-plate-ish at times:

For the sake of documentation we'll be writing a word counter program that will store the count of each encountered word. We'll need to subclass DocumentFragment so let's pull that in to our namespace:

```
>>> from json_document.document import Document, DocumentFragment
```

The Word class is what will represent each word we've encountered. We'll start by keeping it simple, just a inc() method:

```
>>> class Word(DocumentFragment):
...
...     def inc(self):
...         self.value += 1
```

The WordCounter class will be a custom Document that just has the schema. Here we also see that the schema can be defined once in the document class by creating a `document_schema` property. This is convenient when you have one schema and want to make the life of your users easier. The schema defines an object (with a default value of {}). This object can have additional properties (that is, properties not explicitly mentioned in the schema) but each one has to be an integer. If missing the default value of each property is zero. Finally the special `__fragment_cls` schema entry instructs which DocumentFragment sub-class to instantiate:

```
>>> class WordCounter(Document):
...
...     document_schema = {
...         'type': 'object',
...         'default': {},
...         'additionalProperties': {
...             'type': 'integer',
...             'default': 0,
...             '__fragment_cls': Word
...         }
...     }
```

Having done that we can now start using this:

```
>>> doc = WordCounter({})
```

Default values work:

```
>>> doc['json'].value
0
```

As did our custom class declaration:

```
>>> for word in "json is a nice thing to keep your data, json".split():
...     doc[word].inc()
```

Finally the data is saved and we can inspect it or save it later:

```
>>> doc['json'].value
2
```

2.3.2 Value bridge

So we have all the nice features so far, we even have custom fragment classes to keep our code more maintainable and readable. The last thing that was annoying me was the need to use dictionary notation to access my fragments. I wanted to use object traversal notation instead. I ended up writing a lot of properties that were just exposing the fragment in a more natural syntax.

Let's see what it was like:

```
>>> class Config(Document):
...
...     document_schema = {
...         'type': 'object',
...         'default': {},
...         'properties': {
...             'save_on_exit': {
...                 'type': 'bool',
...                 'default': True
...             }
...         }
...     }
...
...     @property
...     def save_on_exit(self):
...         return self['save_on_exit']

>>> conf = Config()
>>> conf.save_on_exit.value
True
```

It worked but was somewhat tedious (I had to repeat the name of the property. It was also annoying if it the property was a simple value (not something more complicated that itself would be having extra methods/state) and I had to type .value all the time.

So I wrote three good decorators that made this easy. They are all in the bridge module:

```
>>> from json_document import bridge
```

We can now improve our Config class with one of them the 'readwrite' bridge:

```
>>> class BetterConfig(Config):
...
...     @bridge.readwrite
...     def save_on_exit(self):
...         ''' documentation on this property '''
```

The intent and code is very clear, it simply allows you to read and write the .value directly, without having the extra lookup on your side. It also gives your JSON document pythonic look and documentation:

```
>>> conf = BetterConfig()
>>> conf.save_on_exit
True
>>> conf.save_on_exit = False
>>> conf.save_on_exit
False
```

If something is not really going to change (say you are only reading a part of a document that is modified by third party program) you can make that explicit in your code by using `bridge.readonly` instead.

2.3.3 Fragment bridge

Fragment bridge is very similar to the value bridge (readonly and readwrite) but instead of returning the value it returns the fragment itself. It allows for more readable code that can still access all the methods and properties that DocumentFragment provides.

I found it useful to document my JSON structure on the python side by mapping larger pieces of the schema to custom classes and putting fragment bridges in the document class.

Let's say you have a person record with first and last name strings:

```
>>> class PersonName(DocumentFragment):
...     """ Person's name """
...
...     @bridge.readwrite
...     def first(self):
...         """ First name """
...
...     @bridge.readwrite
...     def last(self):
...         """ Last name """
...
...     @property
...     def full(self):
...         return "%s %s" % (self.first, self.last)

>>> class Person(Document):
...     """ Person record """
...
...     document_schema = {
...         'type': object,
...         'properties': {
...             'name': {
...                 'type': 'object',
...                 'default': {},
...                 '__fragment_cls': PersonName,
...                 'properties': {
...                     'first': {
...                         'type': 'string'
...                     },
...                     'last': {
...                         'type': 'string'
...                     }
...                 }
...             }
...         }
...     }
...
...     @bridge.fragment
...     def name(self):
...         """ Name data """
```

Uh, that was verbose, the good part is that `after` the bulky class is written we can write lean code using that class. Let's see how this works:

```
>>> john = Person({})
>>> john.name.first = "John"
>>> john.name.last = "Doe"
>>> john.name.full
```

```
'John Doe'
>>> john.value
{'name': {'last': 'Doe', 'first': 'John'}}
```

Did you notice this was a JSON object? Nice eh :-)

That's it

2.4 Code Reference

2.4.1 json_document.document

Document and fragment classes

class `json_document.document.`**`Document`**(*value*, *schema=None*)
Class representing a smart JSON document

A document is also a fragment that wraps the entire value. Is inherits all of its properties. There are two key differences: a document has no parent fragment and it holds the revision counter that is incremented on each modification of the document.

`revision`
Return the revision number of this document.

Each change increments this value by one. You should not really care about the count as sometimes the increments may be not what you expected. It is best to use this to spot difference (if your count is different than mine we're different).

class `json_document.document.`**`DocumentFragment`**(*document*, *parent*, *value*, *item=None*, *schema=None*)
Wrapper around a fragment of a document.

Fragment may wrap a single item (such as None, bool, int, float, string) or to a container (such as list or dict). You can access the value pointed to with the `value` property.

Each fragment is linked to a `parent` fragment and the :attr:'*document* itself. When the parent fragment wraps a list or a dictionary then the index (or key) that this fragment was references as is stored in `item`. Sometimes this linkage becomes broken and a fragment is considered orphaned. Orphaned fragments still allow you to read the value they wrap but since they are no longer associated with any document you cannot set the value anymore.

Fragment is also optionally associated with a schema (typically the relevant part of the document schema). When schema is available you can `validate()` a fragment for correctness. If the schema designates a `default_value` you can `revert_to_default()` to discard the current value in favour of the one from the schema.

`default_value`
Get the default value.

Note: This method will raise SchemaError if the default is not defined in the schema

`default_value_exists`
Returns True if a default value exists for this fragment.

The default value can be accessed with `default_value`. You can also revert the current value to default by calling `revert_to_default()`.

When there is no default value any attempt to use or access it will raise a SchemaError.

document
The document object (the topmost parent document fragment)

is_default
Check if this fragment points to a default value.

Note: A fragment that points to a value equal to the value of the default is **not** considered default. Only fragments that were not assigned a value previously are considered default.

---

is_orphaned
Check if a fragment is orphaned.

Orphaned fragments can occur in this scenario:

```
>>> doc = Document()
>>> doc["foo"] = "value"
>>> foo = doc["foo"]
>>> doc.value = {}
>>> foo.is_orphaned
True
```

That is, when the parent fragment value is overwritten.

item
The index of this fragment in the parent collection.

Item is named somewhat misleadingly. It is the name of the index that was used to access this fragment from the parent fragment. Typically this is the dictionary key name or list index.

parent
The parent fragment (if any)

The document root (typically a `Document` instance) has no parent. If the parent exist then `fragment.parent[fragment.item]` points back to the same value as `fragment` but wrapped in a different instance of DocumentFragment.

revert_to_default()
Discard current value and use defaults from the schema.

@raises TypeError: when default value does not exist Revert the value that this fragment points to to the default value.

schema
Schema associated with this fragment

Schema may be None

This is a read-only property. Schema is automatically provided when a sub-fragment is accessed on a parent fragment (all the way up to the document). To provide schema for your fragments make sure to include them in the `properties` or `items`. Alternatively you can provide `additionalProperties` that will act as a catch-all clause allowing you to define a schema for anything that was not explicitly matched by `properties`.

validate()
Validate the fragment value against the schema

value
Value being wrapped by this document fragment.

Getting reads the value (if not `is_default`) from the document or transparently returns the default values from the schema (if `default_value_exists`).

Setting a value instantiates default values in this or any parent fragment. That is, if the value of this fragment or any of the parent fragments is default (`is_default` returns True), then the default value is copied and used as the effective value instead.

When `is_default` is True setting any value (including the value of `default_value`) will overwrite the value so that `is_default` will return False. If you want to *set the default value* use `revert_to_default()` explicitly.

Setting a value that is different from the current value bumps the revision of the whole document.

class `json_document.document.`**`DocumentPersistence`**(*document*, *storage*, *serializer=None*)
Simple glue layer between document and storage:

```
document <-> serializer <-> storage
```

You can have any number of persistence instances associated with a single document.

`load`()
Load the document from the storage layer

`save`()
Save the document to the storage layer.

The document is only saved if the document was modified since it was last saved. The document revision is non-persistent property (so you cannot use it as a version control system) but as long as the document instance is alive you can optimize saving easily.

2.4.2 Exceptions

exception `json_document.errors.`**`DocumentError`**(*document*, *msg*)
Base class for all Document exceptions.

exception `json_document.errors.`**`DocumentSyntaxError`**(*document*, *error*)
Syntax error in document

exception `json_document.errors.`**`DocumentSchemaError`**(*document*, *error*)
Schema error in document

exception `json_document.errors.`**`OrphanedFragmentError`**(*fragment*)
Exception raised when an orphaned document fragment is being modified.

A fragment becomes orphaned if a saved reference no longer belongs to any document tree. This can happen when one reverts a document fragment to default value while still holding references do elements of that fragment.

exception `json_document.errors.`**`DocumentError`**(*document*, *msg*)
Base class for all Document exceptions.

exception `json_document.errors.`**`DocumentSchemaError`**(*document*, *error*)
Schema error in document

exception `json_document.errors.`**`DocumentSyntaxError`**(*document*, *error*)
Syntax error in document

exception `json_document.errors.`**`OrphanedFragmentError`**(*fragment*)
Exception raised when an orphaned document fragment is being modified.

A fragment becomes orphaned if a saved reference no longer belongs to any document tree. This can happen when one reverts a document fragment to default value while still holding references do elements of that fragment.

2.4.3 json_document.serializers

Document serializer classes

class `json_document.serializers.`**`JSON`**
JSON class encapsulates loading and saving JSON files using simplejson module. It handles 'raw' json without any of the additions specified in the `Document` class.

classmethod **`dump`**(*stream*, *doc*, *human_readable=True*, *sort_keys=False*)
Dump JSON to a stream-like object

Discussion If human_readable is True the serialized stream is meant to be read by humans, it will have newlines, proper indentation and spaces after commas and colons. This option is enabled by default.

If sort_keys is True then resulting JSON object will have sorted keys in all objects. This is useful for predictable format but is not recommended if you want to load-modify-save an existing document without altering it's general structure. This option is not enabled by default.

Return value None

classmethod **`dumps`**(*doc*, *human_readable=True*, *sort_keys=False*)
Dump JSON to a string

Discussion If human_readable is True the serialized value is meant to be read by humans, it will have newlines, proper indentation and spaces after commas and colons. This option is enabled by default.

If sort_keys is True then resulting JSON object will have sorted keys in all objects. This is useful for predictable format but is not recommended if you want to load-modify-save an existing document without altering it's general structure. This option is not enabled by default.

Return value JSON document as string

classmethod **`load`**(*stream*, *retain_order=True*)
Load a JSON document from the specified stream

Discussion The document is read from the stream and parsed as JSON text.

Return value The document loaded from the stream. If retain_order is True then the resulting objects are composed of ordered dictionaries. This mode is slightly slower and consumes more memory but allows one to save the document exactly as it was before (apart from whitespace differences).

Exceptions

JSONDecodeError When the text does not represent a correct JSON document.

classmethod **`loads`**(*text*, *retain_order=True*)
Same as load() but reads data from a string

exception `json_document.serializers.`**`JSONDecodeError`**(*msg*, *doc*, *pos*, *end=None*)
Subclass of ValueError with the following additional properties:

msg: The unformatted error message doc: The JSON document being parsed pos: The start index of doc where parsing failed end: The end index of doc where parsing failed (may be None) lineno: The line corresponding to pos colno: The column corresponding to pos endlineno: The line corresponding to end (may be None) endcolno: The column corresponding to end (may be None)

2.4.4 json_document.storage

Storage classes (for storing serializer output)

class json_document.storage.**FileStorage**(*pathname*, *ignore_missing=False*)
File-based storage class.

This class is used in conjunction with DocumentPersistance to *bind* a Document to a file (via a serializer).

read()
Read all data from the file.

Data is transparently interpreted as UTF-8 encoded Unicode string. If ignore_missing is True and the file does not exist an empty string is returned instead.

write(*data*)
Write the specified data to the file

The data overwrites anything present in the file earlier. If data is an Unicode object it is automatically converted to UTF-8.

class json_document.storage.**IStorage**
Interface for storage classes

read()
Read all data from the storage

write(*data*)
Write data to the storage

2.4.5 json_document.bridge

Collection of decorator methods for accessing document fragments

You want to use those decorators if you are not interested in raw JSON or high-level DocumentFragments (which would require you to access each value via the .value property) but want to offer a pythonic API instead.

json_document.bridge.**fragment**(*func*)
Bridge to a document fragment.

The name of the fragment is identical to to the name of the decorated function. The function is never called, it is only used to obtain the docstring.

This is equivalent to:

```
@property
def foo(self):
    return self['foo']
```

json_document.bridge.**readonly**(*func*)
Read-only bridge to the value of a document fragment.

The name of the fragment is identical to to the name of the decorated function. The function is never called, it is only used to obtain the docstring.

This is equivalent to:

```
@property
def foo(self):
    return self['foo'].value
```

json_document.bridge.**readwrite**(*func*)

Read-write bridge to the value of a document fragment.

The name of the fragment is identical to to the name of the decorated function. The function is never called, it is only used to obtain the docstring.

This is equivalent to:

```
@property
def foo(self):
    return self['foo'].value
```

Followed by:

```
@foo.setter
def foo(self, new_value):
    return self['foo'] = new_value
```

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

j