TraceableDict Documentation

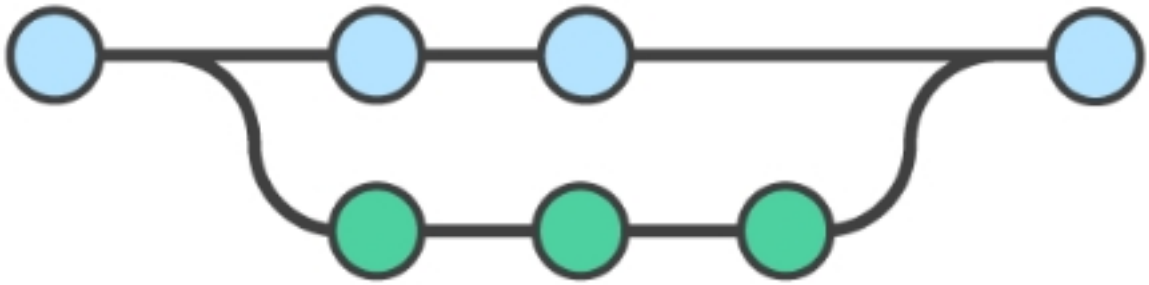
Release 1.0

Shahar Azulay, Rinat Ishak

Oct 17, 2018

Overview

Traceable Python dictionary, that stores change history in an efficient way inside the object.



Shahar Azulay, Rinat Ishak

Usage Examples:

Create a traceable dictionary

```
>>> from traceable_dict import TraceableDict
>>> d1 = {'old_key': 'old_value'}
>>> D1 = TraceableDict(d1)
>>> D1
{'old_key': 'old_value', '__trace__': {}, '__revisions__': []}
>>> D1.revisions
[]
```

Commit the dictionary for the first time

```
>>> D1.has_uncommitted_changes
True
>>>
>>> D1.commit(revision=1)
>>> D1
{'old_key': 'old_value', '__trace__': {}, '__revisions__': [1]}
>>> D1.revisions
[1]
>>> D1.has_uncommitted_changes
False
```

Update the dictionary while tracing the changes

```
>>> D1['new_key'] = 'new_value'
>>> D1.trace
{'_uncommitted_': [(('_root_', 'new_key'), None, '__a__')]}
>>> D1.has_uncommitted_changes
True
>>> D1.commit(revision=2)
>>> D1.trace
{'2': [(('_root_', 'new_key'), None, '__a__')]}
>>> D1.has_uncommitted_changes
```

(continues on next page)

(continued from previous page)

```
False
>>> D1.revisions
[1, 2]
```

Checkout previous revisions

```
>>> D1.as_dict()
{'old_key': 'old_value', 'new_key': 'new_value'}
>>>
>>> D_original = D1.checkout(revision=1)
>>> D_original.as_dict()
{'old_key': 'old_value'}
```

CHAPTER 1

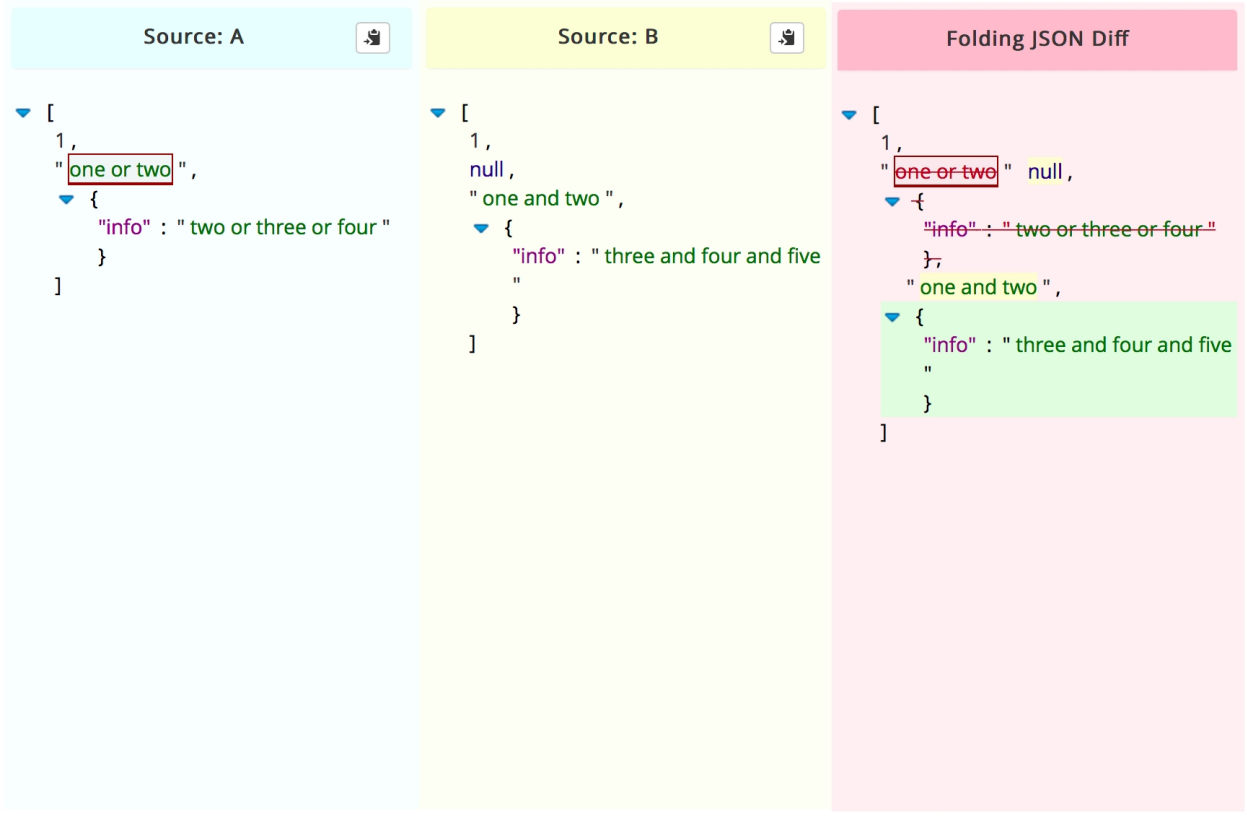
General Concept

Often a Python dictionary object is used to represent a pre-known structured data that captures some state of a system. A non-relational database such as DB is a great example of such use-case, where the BSON-based documents can easily be loaded into a Python dictionary. Those dict-like documents help store complex information, whose structure may change over time, and are highly common in the industry.

In cases where the dictionary or JSON-like structure represents a meaningful state of a system, tracing it changes may be a highly valueble part in the monitoring of the system.

This module implements a traceable Python dictionary, that stores change history in an efficient way inside the object. It allows the user to:

1. **Trace reverions** of the dictionary's content and structure.
2. **Roll the dictionary back** to previously stored values.
3. **Trace the changes** between its different revisions.
4. **Revert** unwanted changes made.
5. **Provide a meaningful id** to the revisions - such as a timestamp, or verison number.
6. More....



[1] tracing the changes in a JSON-like object

CHAPTER 2

The Solution

There are many possible solutions to trace the changes in a dict-like object. The major differences between them is the way in which the trace history is stored.

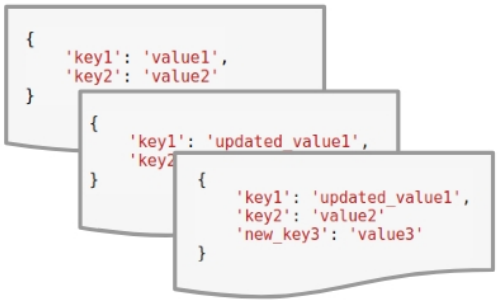
The three main possibilities go back to:

1. **In-Object** solution - where the trace is embedded into the dict-like object itself.
2. **Out-Of-Object** solution - where the trace is stored using some additional attribute of the dict-like object.
3. **Trace by Multiple Objects** solution - where the trace is stored by storing multiple copies of the dict-like object, usually equal to the number of known reivisions.

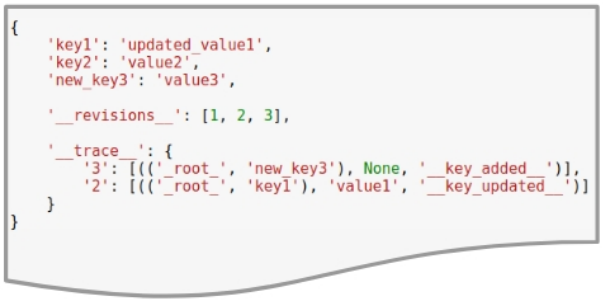
The use of the Out-Of-Object method is not relevant in cases where the object needs to go through serializaion, such as in cases where the object needs to be stored on disk, in a database or in any other non-Python native and consistent form. Therefore, we chose to not address this solution as viable.

We chose to focus our solution to work well for non-relational DBs, which store document JSON-like documents natively. The *Trace by Multiple Objects* solution would force the creation of multiple documents in the DB, possibly resulting in a high memory overhead, if objects are kept in full.

However, such solution would provide quick access time for the latest revision of the document. A possible upgrade of this solution would be to store diffs between document revisions only, but that would possiblt result in a slower accesss time of the latest version.



TRACE BY MULTIPLE OBJECTS



IN-OBJECT TRACE

[1] In-Objecr and Multiple Objects methods for tracing the changes in a JSON-like object

We chose to store the trace *In-Object*. While this method is limited by the max allowed size of the document, and may not be suitable for very large documents, we found it to be the most elegant solution.

The trace is stored as part of the dict-like structure of the document allowing **quick access** to the latest revision, while storing only diffs between revision which results in **lower memory costs**.

CHAPTER 3

Memory Performance

The In-Object trace solution we chose results stores the latest version of the dictionary, and with it two meta-fields that describe the history of the dict-like object:

1. **trace** - capturing diffs between different revisions of the dict over the different revisions.
2. **revision** - capturing the ids of the different revision in which the dict changes.

The space performance is therefore effected directly and linearly by the dict average size, and by the number of revisions, per-key in the dict.

In order to support real world memory restrictions, such as MongoDb maximum document size (16MB), the TraceableDict also support a limited "memory" if needed and can drop old revisions, allowing it to store the latest k-revision only in a cyclic manner.

CHAPTER 4

RunTime Performance

Here are the general asymptotic bounds of expected runtime performance:

1. **as_dict** - Access to the latest dict revision is done in **O(k)**, where k is the number of k
2. **commit** - Assigning a meaningful revision id to all uncommited changes is done in **O(1)**.
3. **revert** - Reverting all uncommited changes is done in **O(1)**.
4. **checkout** - Rolling back to an old revision is done in **O(m + n)** where m is the number of revisions between the working tree and the desired revision, and n is the number of per-key diffs performed between the two revisions.
5. **remove_oldest_revision** - Removing the oldest revision is done in **O(1)**.
6. **log** - Displaying commit logs shows similar performance to *checkout* method.
7. **diff** - Showing changes between revisions shows similar performance to *checkout* method.

CHAPTER 5

Creating the traceable dict for the first time

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'first_key': 'first_value'}
>>>
>>> D1 = TraceableDict(d1)
>>> D1
{'__trace__': {}, '__revisions__': [], 'first_key': 'first_value'}
>>> D1.revisions
[]
>>> D1.has_uncommitted_changes
True
>>>
>>> D1.commit(revision=1)
>>>
>>> D1.has_uncommitted_changes
False
>>> D1.revisions
[1]
```

CHAPTER 6

Updating a single key inside the dictionary

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'old_key': 'old_value'}
>>> D1 = TraceableDict(d1)
>>> D1
{'old_key': 'old_value', '__trace__': {}, '__revisions__': []}
>>>
>>> D1.has_uncommitted_changes
True
>>>
>>> D1.commit(revision=1)
>>> D1
{'old_key': 'old_value', '__trace__': {}, '__revisions__': [1]}
>>> D1.revisions
[1]
>>> D1.has_uncommitted_changes
False
>>> D1['new_key'] = 'new_val'
>>> D1.trace
{'_uncommitted_': [(('_root_', 'new_key'), None, '__a__')]}
>>> D1.has_uncommitted_changes
True
>>> D1.commit(revision=2)
>>> D1.trace
{'2': [(('_root_', 'new_key'), None, '__a__')]}
>>> D1.has_uncommitted_changes
False
>>> D1.revisions
[1, 2]
```

CHAPTER 7

Updating the entire dictionary while tracing the changes

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'old_key': 'old_value'}
>>> D1 = TraceableDict(d1)
>>> D1
{'old_key': 'old_value', '__trace__': {}, '__revisions__': []}
>>>
>>> D1.commit(revision=1)
>>> D1.trace
{}
>>> d2 = {'old_key': 'updated_value', 'new_key': 'new_value'}
>>>
>>> D1 = D1 | d2
>>> D1.as_dict()
{'old_key': 'updated_value', 'new_key': 'new_value'}
>>> D1.trace
{'_uncommitted_': [(('_root_', 'old_key'), 'old_value', '__u__'), (('_root_', 'new_key
↪'), None, '__a__')]}
>>>
>>> D1.commit(revision=2)
>>> D1.trace
{'2': [(('_root_', 'old_key'), 'old_value', '__u__'), (('_root_', 'new_key'), None, '_
↪_a__')]}
>>> D1.has_uncommitted_changes
False
>>> D1.revisions
[1, 2]
```

CHAPTER 8

Reverting un-committed changes to a dictionary

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'old_key': 'old_value'}
>>> D1 = TraceableDict(d1)
>>> D1.commit(revision=1)
>>>
>>> D1['new_key'] = 'new_value'
>>> D1.as_dict()
{'old_key': 'old_value', 'new_key': 'new_value'}
>>> D1.trace
{'_uncommitted_': [(('_root_', 'new_key'), None, '__a__')]}
>>>
>>> D1.revert()
>>>
>>> D1.has_uncommitted_changes
False
>>> D1.as_dict()
{'old_key': 'old_value'}
```

CHAPTER 9

Checkout previous revisions of the dictionary

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'old_key': 'old_value'}
>>> D1 = TraceableDict(d1)
>>> D1.commit(revision=1)
>>>
>>> d2 = {'old_key': 'updated_value', 'new_key': 'new_value'}
>>>
>>> D1 = D1 | d2
>>> D1.as_dict()
{'old_key': 'updated_value', 'new_key': 'new_value'}
>>>
>>> D1.commit(revision=2)
>>> D1.revisions
[1, 2]
>>>
>>> D_original = D1.checkout(revision=1)
>>> D_original.as_dict()
{'old_key': 'old_value'}
```

CHAPTER 10

Displaying the commit logs over the different revisions

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'key1': 'value1'}
>>> D1 = TraceableDict(d1)
>>> D1.commit(revision=1)
>>>
>>> D1['key1'] = 'new_value1'
>>> D1.commit(revision=2)
>>>
>>> log = D1.log(path=('key1',))
changeset:   1
value:       {'key1': 'value1'}


changeset:   2
value:       {'key1': 'new_value1'}


>>> log
{1: {'key1': 'value1'}, 2: {'key1': 'new_value1'}}
```

CHAPTER 11

Show changes between revisions, or latest revision and working tree

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {
... 'key1': 'value1',
... 'key2': 'value2'
... }
>>> D1 = TraceableDict(d1)
>>> D1.commit(revision=1)
>>>
>>> d2 = {
... 'key1': 'new_value1',
... 'key3': 'value3'
... }
>>>
>>> D1 = D1 | d2
>>> D1.commit(revision=2)
>>>
>>> diff = D1.diff(revision=2)
>>> diff
{'key3': '+++value3', 'key2': '---value2', 'key1': '---value1 +++new_value1'}
```

CHAPTER 12

Removing the oldest revision of the traceable dict

This option allows the user to contol the amount of revisions stored in the traceable-dict object, by trimming the tail of the trace stored in the traceable-dict. The oldest revision is cleared out and cannot be returned to again.

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'old_key': 'old_value'}
>>> d2 = {'old_key': 'new_value'}
>>> d3 = {'old_key': 'even_newer_value'}
>>>
>>> D1 = TraceableDict(d1)
>>> D1.commit(revision=1)
>>>
>>> D1 = D1 | d2
>>> D1.commit(revision=2)
>>>
>>> D1 = D1 | d3
>>> D1.commit(revision=3)
>>>
>>> D1.revisions
[1, 2, 3]
>>> D1.remove_oldest_revision()
>>> D1. revisions
[2, 3]
```