vivodict Documentation

Release 0.3.1

Adamos Kyriakou

Oct 27, 2020

Contents

1 vivodict **3**
- 1.1 Features . . . 3
- 1.2 Motivation . . . 3
- 1.3 Basic Usage . . . 3
- 1.4 Convenience Functions . . . 4

2 Installation **7**
- 2.1 Stable release . . . 7
- 2.2 From sources . . . 7

3 History **9**
- 3.1 0.3.1 (2017-07-23) . . . 9
- 3.2 0.3.0 (2017-07-23) . . . 9
- 3.3 0.2.0 (2017-07-23) . . . 9
- 3.4 0.1.1 (2017-07-23) . . . 9
- 3.5 0.1.0 (2017-07-23) . . . 9

Contents:

CHAPTER 1

vivodict

This package provides a simple implementation of an auto-vivified Python `dict`, i.e., a dictionary where accessing a missing key doesn't raise the standard `KeyError` exception but instead implicitly creates and returns an empty auto-vivified `dict` under that key.

1.1 Features

- Auto-vivified `VivoDict` class derived from the standard Python `dict` class (no third-party dependencies).
- Auto-vivification of arbitrarily-nested `dict` objects.
- Convenience methods for `flatten`, `replace`, and `apply` operations.
- Free software: MIT license
- Documentation: https://vivodict.readthedocs.io.

1.2 Motivation

My primary motivation for developing this package is because it contained a piece of code I kept copy-pasting like a bloody caveman between projects.

My typical use-cases for this code include:

- Wrap the decoded JSON `dict` from crummy APIs without a schema that just decide to drop keys for which the values are `null` resulting in code with nested `if "key" in result:`. This allowed me to either retrieve the value if it was there or at least arriving at an empty `dict` which evaluates to `False` when mapping their half-formed data to my own data-structures.
- Create arbitrarily-nested dictionaries of code stats that I can keep organized as I like while using in the code and then quickly `flatten` to a Graphite compatible format prior to posting them to ... well Graphite.

1.3 Basic Usage

This would be the typical Python `dict` behaviour when accessing a missing key:

```
>>> d = {"a": 1, "b": 2}
>>> d["a"]
1
```

```
>>> d["missing"]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-3-d4f58b57b715> in <module>()
----> 1 d["missing"]

KeyError: 'missing'
```

While if we were using a `VivoDict`, then upon accessing a missing key we would be provided with an implicitly created empty `VivoDict` as such:

```
>>> from vivodict import VivoDict
>>> d = VivoDict.vivify({"a": 1, "b": 2})
>>> d["a"]
1
>>> d["missing"]
{}
```

Note: Note that instantiation above is not performed simply by passing an existing `dict` to `VivoDict` but instead through the `vivify` class method which can recursively convert any arbitrarily-nested `dict` to a `VivoDict`.

Now, while the above doesn't seem to offer anything a simple `try-except` or a `if "key" not in d` wouldn't offer, the `VivoDict` becomes useful when dealing with arbitrarily nested dictionaries where there may be several levels of missing keys. For example:

```
>>> from vivodict import VivoDict
>>> d = VivoDict({"a": 1, "b": {"c": 2}, "d": {"e": {"f": 3}}})
>>> d["a"]
1
>>> d["b"]["c"]
2
>>> d["d"]["e"]["f"]
3
>>> d["i"]["am"]["missing"]["eh"] = 4
>>> d
{'a': 1,
'b': {'c': 2},
'd': {'e': {'f': 3}},
'i': {'am': {'missing': {'eh': 4}}}}
```

So, as can be seen, having auto-vivification allows one to nest keys and values to whatever degree.

Warning: The primary caveat of the above functionality is that `VivoDict` are very forgiving when it comes to typos which can leads to weird errors. A mistyped key will simply create a new `VivoDict` and will allow you to go down some rabbithole of erroneously typed keys your linter won't get you out of.

1.4 Convenience Functions

In addition to the above, a few basic convenience methods have been built into the `VivoDict` class, mostly cause they make my life easier and lazier.

1.4.1 flatten

As I mentioned prior one of my typical use-cases for `vivodict` is using it to store nested metrics which I then post to Graphite via simple HTTP requests.

Graphite, however, bases its structure on `.` delimited names where anything preceding a `.` is considered to be a metric folder with the last token being the metric itself.

Thus, I needed a quick way to flatten a nested `dict` into a Graphite compatible version.

The `flatten` method does exactly that:

```
>>> d = VivoDict.vivify({"a": 1, "b": {"c": 2}, "d": {"e": {"f": 3}}})
>>> d.flatten()
{'a': 1, 'b.c': 2, 'd.e.f': 3}
```

1.4.2 replace

Following the same premise as with `flatten` I needed to quickly 'reset' my metrics back to 0 between posting cycles.

Hence, `replace` will replace all 'leaf' node values in what is essentially a tree with a given `value`:

```
>>> d = VivoDict.vivify({"a": 1, "b": {"c": 2}, "d": {"e": {"f": 3}}})
>>> d.replace(replace_with=0)
>>> d
{'a': 0, 'b': {'c': 0}, 'd': {'e': {'f': 0}}}
```

Warning: As you may have noticed from the above snippet, the `replace` method performs an **in-place** replacement instead of returning a copy of the original `VivoDict` with replaced values.

Should you need to maintain an original copy I'd suggest you use the `copy` package and its `deepcopy` function (cause Python passes by reference) as such:

```
>>> import copy
>>> original = VivoDict.vivify({"a": 1, "b": {"c": 2}, "d": {"e": {"f": 3}}})
>>> modified = copy.deepcopy(original)
>>> modified.replace(replace_with=0)
>>> original
{'a': 1, 'b': {'c': 2}, 'd': {'e': {'f': 3}}}
>>> modified
{'a': 0, 'b': {'c': 0}, 'd': {'e': {'f': 0}}}
```

1.4.3 apply

Lastly, I often had to modify all values through a given function, typically divide them by a number of observation for average metrics which can be easily done through the `apply` method which can take any `callable` as an argument and replace the original value with its return-value:

```
>>> d = VivoDict.vivify({"a": 1, "b": {"c": 2}, "d": {"e": {"f": 3}}})
>>> def double(value):
>>>     return value * 2
>>> d.apply(double)
>>> d
```

```
{'a': 2, 'b': {'c': 4}, 'd': {'e': {'f': 6}}}
>>> d.apply(lambda value: value / 2)
{'a': 1, 'b': {'c': 2}, 'd': {'e': {'f': 3}}}
```

Warning: Much like `replace`, the `apply` method replaces values **in-place**.

CHAPTER 2

Installation

2.1 Stable release

To install vivodict, run this command in your terminal:

```
$ pip install vivodict
```

This is the preferred method to install vivodict, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for vivodict can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/somada141/vivodict
```

Or download the tarball:

```
$ curl  -OL https://github.com/somada141/vivodict/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

History

3.1 0.3.1 (2017-07-23)

- README.rst: Fixed minor formatting typo.

3.2 0.3.0 (2017-07-23)

- Cleanup the docos and removed a bunch of the unnecessary stuff.

3.3 0.2.0 (2017-07-23)

- Added more unit-tests and improved docstrings.

3.4 0.1.1 (2017-07-23)

- Fixed issues with the Python dependencies.

3.5 0.1.0 (2017-07-23)

- First release on PyPI.