sndict Documentation

Release 0.1.2

Jason Phang

Apr 16, 2019

Contents

CHAPTER 1

Home

Nested Extensions to Python dictionaries

- Free software: MIT license
- Documentation: https://sndict.readthedocs.io
- Code: https://github.com/zphang/sndict

1.1 Introduction

This module provides extensions to `dicts` in the python standard library, providing fast and clean manipulation of nested dictionary structures. This module exposes two new `dict`-types:

- `NestedDict`/`ndict`: A light-weight wrapper for `dict` s that provides additional functionality for operations on nested dictionary structures.
- `StructuredNestedDict`/`sndict`: A heavy-weight data `dict` -based structure for operating on hierarchical data with rich functionality for filtering and transformation across nested levels.

Both implementations are use `OrderedDict` s under the hood.

No additional dependencies are required.

1.2 Features

- **`NestedDict/ndict`:**
  - Iterating over flattened keys and values

  - Nested getting/setting operations
  - Applicable to dictionaries of arbitrary and unbalanced depth

- **`StructuredNestedDict/sndict:`**
  - `flatten`/`stratify`/`rearrange` methods allow for powerful and rich operations across different levels of hierarchy
  - Nested getting/setting operations, including intelligent filtering via `ix`
  - Convenient data inspection via `dim`, `unique_keys`, etc

CHAPTER 2

Contents:

2.1 Home

Nested Extensions to Python dictionaries

- Free software: MIT license
- Documentation: https://sndict.readthedocs.io
- Code: https://github.com/zphang/sndict

2.1.1 Introduction

This module provides extensions to `dicts` in the python standard library, providing fast and clean manipulation of nested dictionary structures. This module exposes two new `dict`-types:

- `NestedDict`/`ndict`: A light-weight wrapper for `dict` s that provides additional functionality for operations on nested dictionary structures.
- `StructuredNestedDict`/`sndict`: A heavy-weight data `dict` -based structure for operating on hierarchical data with rich functionality for filtering and transformation across nested levels.

Both implementations are use `OrderedDict` s under the hood.

No additional dependencies are required.

2.1.2 Features

- **`NestedDict`/`ndict`:**

  - Iterating over flattened keys and values
  - Nested getting/setting operations
  - Applicable to dictionaries of arbitrary and unbalanced depth

- **`StructuredNestedDict/sndict:`**
  - `flatten`/`stratify`/`rearrange` methods allow for powerful and rich operations across different levels of hierarchy
  - Nested getting/setting operations, including intelligent filtering via `ix`
  - Convenient data inspection via `dim`, `unique_keys`, etc

2.2 Examples

2.2.1 Navigating Directory Structures with NestedDict

We can obtain the directory tree of the `sndict` repo like so:

“‘python import os import sndict

directory_tree = sndict.app.directory_tree(os.path.join(os.path.dirname(sndict.__file__), “..”)) “‘

Unfortunately, this includes various files/folders that we don’t want, for example the .git folder docs/build folders. We can just delete those:

`python del directory_tree[".git"] del directory_tree["docs"]["build"] `

This still leaves us with “.pyc” files. We can remove them using `filter_values`. We can then calculate the file-size of every file in the repo, and print it as a readable tree-string.

“‘python print directory_tree

.filter_values(lambda _: “.pyc” in _, filter_out=True).map_values(lambda _: os.stat(_).st_size).to_tree_string()

Output: # # .gitignore: 49 # .travis.yml: 1057 # AUTHORS.rst: 97 # CONTRIBUTING.rst: 3216 # HISTORY.rst: 275 # LICENSE: 1071 # MANIFEST.in: 264 # Makefile: 2291 # README.rst: 1920 # docs: # ‘-Makefile: 6762 # ‘-app.rst: 67 # ‘-authors.rst: 28 # ‘-conf.py: 8491 # ‘-contributing.rst: 33 # ‘-examples.rst: 895 # ‘-history.rst: 28 # ‘-index.rst: 235 # ‘-installation.rst: 1099 # ‘-make.bat: 6459 # ‘-ndict.rst: 253 # ‘-readme.rst: 27 # ‘-sndict.rst: 508 # ‘-usage.rst: 131 # requirements_dev.txt: 145 # setup.cfg: 339 # setup.py: 1529 # sndict: # ‘-__init__.py: 207 # ‘-app.py: 754 # ‘-exceptions.py: 41 # ‘-nesteddict.py: 13783 # ‘-shared.py: 535 # ‘-structurednesteddict.py: 35819 # ‘-utils.py: 6472 # tests: # ‘-__init__.py: 0 # ‘-test_nesteddict.py: 4111 # ‘-test_shared.py: 456 # ‘-test_structurednesteddict.py: 10624 # tox.ini: 395 # travis_pypi_setup.py: 3751

“‘

2.3 NestedDict / ndict

`NestedDict`/`ndict` s are designed to be light-weight wrappers around existing nested dictionaries, providing additional methods for nested operations.

class `sndict.nesteddict.`**`NestedDict`**(**args*, ***kwargs*)

Extension of OrderedDict that exposes operations on nested dicts

args kwargs

convert (*dict_type=None*, *sort_keys=True*, *key=None*, *reverse=False*)
Convert all nested dictionaries to desired type.

dict_type: ['ndict', 'dict', 'odict'] Dict-type in string format

sort_keys: bool Whether to sort keys

key: function, optional Key function

reverse: bool, optional Whether to sort in reverse

dict, OrderedDict or NestedDict

filter_values (*criteria*, *filter_out=False*)
Filter NestedDict values by criteria.

The criteria used in the following ways, based on type:

1. slice(None): Keep all
2. function: Keep if function(key) is True
3. list, set: Keep if key in list/set
4. other: Keep if key==other

criteria: See above Filter based on criteria

filter_out: bool Whether to filter in or out

NestedDict

flatten (*max_depth=None*)

Iterate over a flattened NestedDict. For each value, keys along the DFS are accumulated as a tuple

max_depth: int, optional Maximum depth to flatten by, i.e. the maximum key-tuple length

list List of (key-tuple, value) pairs

flatten_keys (*max_depth=None*)
Iterate over a flattened NestedDict, and expose only keys. Keys along the DFS are accumulated as a tuple

max_depth: int, optional Maximum depth to flatten by, i.e. the maximum key-tuple length

list List of key-tuples

flatten_values (*max_depth=None*)
Iterate over a flattened NestedDict, and expose only values.

max_depth: int, optional Maximum depth to flatten by

list List of values

classmethod from_flat (*data*, *dict_type='ndict'*)
Initialize from a dict keyed by tuples Each tuple-element is taken as a key for each level in the NestedDict

data: dict, list Dictionary keyed by tuples

dict_type: ['ndict', 'dict', 'odict'] Dict-type in string format, for initializing dicts at depth if they don't exist yet

NestedDict

has_nested_key(*key_list*)
Check if nested keys are valid

key_list: list List of keys, one for each dict depth

bool

iterflatten(*max_depth=None*)

Iterate over a flattened NestedDict. For each value, keys along the DFS are accumulated as a tuple

max_depth: int, optional Maximum depth to flatten by, i.e. the maximum key-tuple length

iterator Iterator of (key-tuple, value) pairs

iterflatten_keys(*max_depth=None*)
Iterate over a flattened NestedDict, and expose only keys. Keys along the DFS are accumulated as a tuple

max_depth: int, optional Maximum depth to flatten by, i.e. the maximum key-tuple length

iterator Iterator of key-tuples

iterflatten_values(*max_depth=None*)
Iterate over a flattened NestedDict, and expose only values.

max_depth: int, optional Maximum depth to flatten by

iterator Iterator of values

ix
Indexer that allows for indexing by nested key list e.g.

my_ndict.ix[“key1”, “key2”, “key3”]

Also supports:

my_ndict.ix[“key1”]

If behavior is ambiguous, use ndict.ixkeys[key_list] and ndict[key] directly instead

Indexable

ixkeys
Indexer that allows for indexing by nested key list e.g.

my_ndict.ix[“key1”, “key2”, “key3”]

Note that for a single key, a tuple/list needs to be provided, e.g.

my_ndict.ix[“key1”,]

Indexable

map_values(*val_func*)
Apply transformations to keys and values

val_func: function Function to transform values

NestedDict

nested_get(*key_list*)
Get value at depth

key_list: list List of keys, one for each dict depth

obj

nested_set(*key_list*, *value*, *dict_type='ndict'*)
Set a value within nested dicts, creating dicts at depth if they don't exist yet

key_list: list List of keys, one for each dict depth

value: object Value to set nested

dict_type: ['ndict', 'dict', 'odict'] Dict-type in string format, for initializing dicts at depth if they don't exist yet

nested_setdefault(*key_list*, *default=None*, *dict_type='ndict'*)
Nested version of dict.setdefault, where a value is set only if it doesn't already exist, creating dicts at depth if they don't exist yet

key_list: list List of keys, one for each dict depth

default: object Value to set nested

dict_type: ['ndict', 'dict', 'odict'] Dict-type in string format, for initializing dicts at depth if they don't exist yet

nested_update(*other_dict*)
Nested version of dict.update. Changes NestedDict in-place

other_dict: dict dict to update by.

sort_nested_keys(*key=None*, *reverse=False*)
Sort keys in every nested dictionary

key: function, optional Key function

reverse: bool, optional Whether to sort in reverse

NestedDict

to_tree_string(*indent=' - '*, *key_mode='str'*, *val_mode='str'*)
Returns structure of NestedDict in tree format string

indent: str Indentation string for levels

key_mode: "type", "str" or "repr" How to serialize key

val_mode: "type", "str" or "repr" How to serialize terminal value

str

2.4 StructuredNestedDict / sndict

`StructuredNestedDict`/`sndict` s are heavy-weight data structures hierarchical of `dict` s. Unlike the light-weight `ndict`, `sndict` transforms all underlying `dict` s to `sndict` s and enforces a fixed dictionary depth internally.

In turn, these constraints allow for more powerful operations to be performed across the different levels of hierarchy.

class sndict.structurednesteddict.**StructuredNestedDict**(**args*, ***kwargs*)
Extension of OrderedDict that exposes advanced operations on nested dicts of fixed depth

data: dict, or list Nested dictionary

levels: int Number of levels

level_names: list List of level names

convert (*dict_type=None*)
Convert all nested dictionaries to desired type.

dict_type: ['sndict', 'ndict', 'dict', 'odict'] Dict-type in string format

dict, OrderedDict or NestedDict

dim
Dimensions of whole StructuredNestedDict, up to defined level

tuple: Tuple of widths of nested dictionaries, one per level

dim_dict
Dimensions of whole StructuredNestedDict as dict, up to defined level

dict Dimensions keyed by level name

filter_key (*criteria_ls*, *filter_out=False*, *drop_empty=False*)
Filter StructuredNestedDict by criteria.

The criteria used in the following ways, based on type:

1. slice(None): Keep all
2. function: Keep if function(key) is True
3. list, set: Keep if key in list/set
4. other: Keep if key==other

criteria_ls: list or dict Filter based on criteria

filter_out: bool Whether to filter in or out

drop_empty: Whether to drop empty nested dictionaries (nested dictionaries with all elements filtered out)

StructuredNestedDict

filter_values (*criteria*, *filter_out=False*, *level=None*, *drop_empty=False*)
Filter StructuredNestedDict values by criteria.

The criteria used in the following ways, based on type:

1. slice(None): Keep all
2. function: Keep if function(key) is True
3. list, set: Keep if key in list/set
4. other: Keep if key==other

criteria: See above Filter based on criteria

filter_out: bool Whether to filter in or out

drop_empty: Whether to drop empty nested dictionaries (nested dictionaries with all elements filtered out)

StructuredNestedDict

flatten (*levels=-1*, *named=True*, *flattened_name=None*)
Returns an StructuredNestedDict with multiple levels flattened

WARNING: If there are empty dictionaries within the levels being flattened, their keys will be lost. This is consistent with the logic of flattening - those dictionaries contain no values for a (level-tuple)-keyed dictionary.

Note: .flatten(levels=0) does nothing, .flatten(levels=1) compresses 1 level (i.e. keys will be 2-ples) .flatten(levels=-1) compresses all levels

levels: int, default=1 Number of levels to flatten by. Defaults to flattening one level

named: bool Whether output key-tuples are namedtuples

flattened_name: str Name of new flattened level. Defaults to original level names joined by "___"

StructuredNestedDict

flatten_keys(*levels=-1*, *named=True*)

Returns an list with of keys of flattened dict

WARNING: If there are empty dictionaries within the levels being flattened, their keys will be lost. This is consistent with the logic of flattening - those dictionaries contain no values for a (level-tuple)-keyed dictionary.

Note: .flatten(levels=0) does nothing, .flatten(levels=1) compresses 1 level (i.e. keys will be 2-ples) .flatten(levels=-1) compresses all levels

levels: int, default=1 Number of levels to flatten by. Defaults to flattening one level

named: bool Whether output key-tuples are namedtuples

list

flatten_values(*levels=-1*)

Returns an list with of values of flattened dict

WARNING: If there are empty dictionaries within the levels being flattened, their keys will be lost. This is consistent with the logic of flattening - those dictionaries contain no values for a (level-tuple)-keyed dictionary.

Note: .flatten(levels=0) does nothing, .flatten(levels=1) compresses 1 level (i.e. keys will be 2-ples) .flatten(levels=-1) compresses all levels

levels: int, default=-1 Number of levels to flatten by. Defaults to flattening all levels.

list

get_named_tuple(*levels*)

Get namedtuple class for named keys

levels: Number of levels to construct named keys for

class

classmethod groupby(*data*, *by*, *levels=None*, *level_names=None*)

Initialize by grouping elements from list

data: list or dictionary List or dictionary to group by

by: function Function applied to list elements to form key elements

levels: int Number of levels

level_names: list List of level names

StructuredNestedDict

has_nested_key(*key_list*)
Check if nested keys are valid

key_list: list List of keys, one for each dict depth

bool

iterflatten(*levels=-1*, *named=True*)
Returns an iterator with multiple levels flattened

WARNING: If there are empty dictionaries within the levels being flattened, their keys will be lost. This is consistent with the logic of flattening - those dictionaries contain no values for a (level-tuple)-keyed dictionary.

Note: .flatten(levels=0) does nothing, .flatten(levels=1) compresses 1 level (i.e. keys will be 2-ples) .flatten(levels=-1) compresses all levels

levels: int, default=1 Number of levels to flatten by. Defaults to flattening one level

named: bool Whether output key-tuples are namedtuples

StructuredNestedDict

iterflatten_keys(*levels=-1*, *named=True*)
Returns an iterator with of keys of flattened dict

WARNING: If there are empty dictionaries within the levels being flattened, their keys will be lost. This is consistent with the logic of flattening - those dictionaries contain no values for a (level-tuple)-keyed dictionary.

Note: .flatten(levels=0) does nothing, .flatten(levels=1) compresses 1 level (i.e. keys will be 2-ples) .flatten(levels=-1) compresses all levels

levels: int, default=1 Number of levels to flatten by. Defaults to flattening one level

named: bool Whether output key-tuples are namedtuples

list

iterflatten_values(*levels=-1*)
Returns an iterator with of values of flattened dict

WARNING: If there are empty dictionaries within the levels being flattened, their keys will be lost. This is consistent with the logic of flattening - those dictionaries contain no values for a (level-tuple)-keyed dictionary.

Note: .flatten(levels=0) does nothing, .flatten(levels=1) compresses 1 level (i.e. keys will be 2-ples) .flatten(levels=-1) compresses all levels

levels: int, default=-1 Number of levels to flatten by. Defaults to flattening all levels.

list

ix
Indexer that allows for indexing by nested key/criteria list e.g.

my_ndict.ix[“key1”, “key2”, “key3”]

or with criteria such as

my_sndict.ix[“key1”, :, lambda _: “3” in _]

Also supports:

my_sndict.ix[“key1”]

If behavior is ambiguous, use sndict.ixkeys[key_list] and sndict[key] directly instead

Indexable

ixkeys
Indexer that allows for indexing by nested key list e.g.

my_ndict.ix[“key1”, “key2”, “key3”]

or with criteria such as

my_sndict.ix[“key1”, :, lambda _: “3” in _]

Note that for a single key, a tuple/list needs to be provided, e.g.

my_sndict.ix[“key1”,]

Indexable

level_names
Names of levels. Defaults to [“level0”, “level1”, . . .] is no names are provided

list

levels
Number of levels

int

map(*key_func=None*, *val_func=None*, *at_level=-1*, *warn=False*)
Apply transformations to keys and values

key_func: function, optional Function to transform keys. Defaults to identity.

val_func: function, optional Function to transform values. Defaults to identity.

at_level: int Level to transform keys at

warn: bool Warn if dimensions of dictionary have been changed

StructuredNestedDict

map_keys(*key_func*, *at_level=-1*)
Apply transformations to keys and values

key_func: function Function to transform keys

at_level: int Level to transform keys at

StructuredNestedDict

map_values(*val_func*, *at_level=-1*)
Apply transformations to keys and values

val_func: function Function to transform values

at_level: int Level to transform values at

StructuredNestedDict

nested_get(*key_list*)
Get value at depth

key_list: list List of keys, one for each dict depth

obj

nested_set(*key_list*, *value*)
Set a value within nested dicts, creating StructuredNestedDict at depth if they don't exist yet

Note: Only allowed to set up to level of StructuredNestedDict

key_list: list List of keys, one for each dict depth

value: object Value to set nested

nested_setdefault(*key_list*, *default=None*)
Nested version of dict.setdefault. Set a value within nested dicts, creating StructuredNestedDict at depth if they don't exist yet

Note: Only allowed to set up to level of StructuredNestedDict

key_list: list List of keys, one for each dict depth

default: object, optional Value to set nested

rearrange(*level_ls=None*, *level_name_ls=None*)
Rearrange levels of StructuredNestedDict Only supply either level_ls or level_name_ls.

Note: Whether supplying level_ls or level_name_ls, the supplied list must cover all levels contiguously from the start. I.e.

level_ls=[2, 0, 1, 3]

is valid but

level_ls=[2, 0]

is not.

level_ls: list List of level ints. If level_ls contains level_names instead, the arguments is passed on to level_name_ls

level_name_ls: list List of level names

StructuredNestedDict

replace_data(*data*)
Return new StructuredNestedDict with different data but same metadata

data: dict, or list Nested dictionary

StructuredNestedDict

replace_metadata(***kwargs*)
Return new StructuredNestedDict with different metadata but same data

levels: int Number of nested levels that StructuredNestedDict will work on

level_names: list List of level names

StructuredNestedDict

sort_keys(*cmp=None*, *key=None*, *reverse=False*)
Sort keys of StructuredNestedDict (top-level only)

cmp: function, optional Comparator function

key: function, optional Key function

reverse: bool, optional Whether to sort in reverse

StructuredNestedDict

sort_values(*key=None*, *reverse=False*)
Sort values of StructuredNestedDict (top-level only)

key: function Key function

reverse: bool Whether to sort in reverse

StructuredNestedDict

stratify(*levels=None*, *stratified_names=None*)

Increases depth (nests) of StructuredNestedDict by splitting up keys in the top-most level

levels: int, default=None Number of levels to stratify by. Defaults to length of first key. Note: levels must be <= length of all top-level keys

stratified_names: default=None Names of newly created stratified levels. Must be same length as levels.

StructuredNestedDict

swap_levels(*level_a*, *level_b*)
Swap two levels in a StructuredNestedDict

Unlike sndict.rearrange, there's no need to be contiguous

level_a: int or str level or level_name

level_b: int or str level or level_name

StructuredNestedDict

to_tree_string(*indent=' - '*, *key_mode='str'*, *val_mode='type'*)
Returns structure of NestedDict in tree format string

indent: str Indentation string for levels

key_mode: "type", "str" or "repr" How to serialize key

val_mode: "type", "str" or "repr" How to serialize terminal value

str

unique_keys(*named=False*, *sort_keys=True*)
Returns the unique keys in each level of the dictionary

named: bool If True, return OrderedDict of list of keys of each level. If False, return a list of list of keys.

sort_keys: bool Whether to sort each list of keys

list or dict

2.5 Applications

sndict.app.**directory_tree**(*base_path*)
Compute directory tree as NestedDict

base_path: Starting path directory Tree

NestedDict

2.6 Installation

2.6.1 Stable release

To install sndict, run this command in your terminal:

```
$ pip install sndict
```

This is the preferred method to install sndict, 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.

2.6.2 From sources

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

You can either clone the public repository:

```
$ git clone git://github.com/zphang/sndict
```

Or download the tarball:

```
$ curl  -OL https://github.com/zphang/sndict/tarball/master
```

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

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

2.7 Usage

To use sndict in a project:

```
import sndict as snd
```

The two primary classes:

```
snd.ndict
snd.sndict
```

2.8 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.8.1 Types of Contributions

2.8.1.1 Report Bugs

Report bugs at https://github.com/zphang/sndict/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

2.8.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

2.8.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

2.8.1.4 Write Documentation

sndict could always use more documentation, whether as part of the official sndict docs, in docstrings, or even on the web in blog posts, articles, and such.

2.8.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/zphang/sndict/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.8.2 Get Started!

Ready to contribute? Here's how to set up *sndict* for local development.

1. Fork the *sndict* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/sndict.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv sndict
$ cd sndict/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 sndict tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

2.8.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/zphang/sndict/pull_requests and make sure that the tests pass for all supported Python versions.

2.8.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_sndict
```

2.9 Credits

2.9.1 Development Lead

- Jason Phang <email@jasonphang.com>

2.10 History

2.10.1 0.1.1 (2017-03-15)

- **Minor functionality update:**
  - *unique_keys*
  - *to_tree_string*

  - Improved filtering/indexing
  - Documentation

2.10.2 0.1.0 (2017-03-14)

- Initial Version with basic functionality

Python Module Index

S

Index