
puffbird: Handling puffy dataframes

Release 0.0.1

Matthias Christenson

Oct 18, 2021

CONTENTS

1	User Guide	3
1.1	Installation	3
1.2	Quick Start	3
1.3	Tutorials	4
1.3.1	Philosophy	4
1.3.2	Creating long dataframes from <i>puffy</i> tables	4
2	API Reference	15
2.1	puffbird.puffy_to_long	15
2.2	puffbird.FrameEngine	17
2.2.1	puffbird.FrameEngine.cols	19
2.2.2	puffbird.FrameEngine.cols_rename	19
2.2.3	puffbird.FrameEngine.datacols	19
2.2.4	puffbird.FrameEngine.datacols_rename	19
2.2.5	puffbird.FrameEngine.indexcols	19
2.2.6	puffbird.FrameEngine.indexcols_rename	19
2.2.7	puffbird.FrameEngine.table	19
2.2.8	puffbird.FrameEngine.apply	20
2.2.9	puffbird.FrameEngine.col_apply	21
2.2.10	puffbird.FrameEngine.drop	21
2.2.11	puffbird.FrameEngine.expand_col	21
2.2.12	puffbird.FrameEngine.multid_pivot	22
2.2.13	puffbird.FrameEngine.rename	22
2.2.14	puffbird.FrameEngine.to_long	22
2.2.15	puffbird.FrameEngine.to_puffy	24
2.3	puffbird.CallableContainer	25
2.3.1	puffbird.CallableContainer.__call__	25
2.3.2	puffbird.CallableContainer.add	25
3	Development	27
3.1	Contributing	27
3.1.1	Types of Contributions	27
3.1.2	Get Started!	28
3.1.3	Pull Request Guidelines	29
3.1.4	Tips	29
3.1.5	Deploying	29
3.2	Style Guide	29
3.2.1	Patterns	29
3.2.2	String formatting	30
3.2.3	Imports (aim for absolute)	31

3.3	Authors	31
3.3.1	Development Lead	31
3.3.2	Contributors	32
4	Release notes	33
4.1	Version 0.0.0	33
4.1.1	Version 0.0.1 (May 15, 2020)	33
	Python Module Index	35
	Index	37

Date: Oct 18, 2021 **Version:** 0.0.1

Useful links: [Binary Installers](#) | [Source Repository](#) | [Issues & Ideas](#)

puffbird is an open source, MIT-licensed library providing an extension to handling *puffy* *pandas* DataFrame objects. For creating the documentation for *puffbird*, I used pandas documentation style.

User Guide

[To the reference guide](#)

{{ header }}

1.1 Installation

Puffbird can be installed via pip from PyPI:

```
pip install puffbird
```

You can also clone the git repository and install the package from source.

1.2 Quick Start

The main functionality that *puffbird* adds to *pandas* is the ability to easily “**explode**” “*puffy*” tables:

```
In [1]: import pandas as pd

In [2]: import puffbird as pb

In [3]: df = pd.DataFrame({
...:     'a': [[1,2,3], [4,5,6,7], [3,4,5]],
...:     'b': [{'c':['asdf'], 'd':['ret']}, {'d':['r']}, {'c':['ff']}]},
...:     })
...:

In [4]: df
Out[4]:
```

	a	b
0	[1, 2, 3]	{'c': ['asdf'], 'd': ['ret']}
1	[4, 5, 6, 7]	{'d': ['r']}
2	[3, 4, 5]	{'c': ['ff']}

As you can see, this dataframe is “*puffy*”, it has various non-hashable object types that can be iterated over. To quickly create a *long-format DataFrame*, we can use the `puffy_to_long` function:

```
In [5]: long_df = pb.puffy_to_long(df)

In [6]: long_df
Out[6]:
```

	index_level0	a_level0	a	b_level0	b_level1	b
0	0	0	1.0	c	0	asdf
1	0	0	1.0	d	0	ret
2	0	1	2.0	c	0	asdf

(continues on next page)

(continued from previous page)

3	0	1	2.0	d	0	ret
4	0	2	3.0	c	0	asdf
5	0	2	3.0	d	0	ret
6	1	0	4.0	d	0	r
7	1	1	5.0	d	0	r
8	1	2	6.0	d	0	r
9	1	3	7.0	d	0	r
10	2	0	3.0	c	0	ff
11	2	1	4.0	c	0	ff
12	2	2	5.0	c	0	ff

1.3 Tutorials

The different tutorials are listed below:

- *Creating long dataframe from puffry tables.*

1.3.1 Philosophy

TODO

1.3.2 Creating long dataframes from *puffry* tables

This tutorial will show the basic features of using the `puffbird.puffry_to_long` method

```
[1]: import numpy as np
import pandas as pd
import puffbird as pb
```

A weirdly complex table

First, we will create a *puffry* dataframe as an example:

```
[2]: df = pd.DataFrame({
    # a and c have the same data repeated three times
    # b is just a bunch of numpy arrays of the same shapes
    # d is also just a bunch of numpy arrays of different shapes
    # e contains various pandas DataFrames with the same column structures
    #   and the same index format.
    # f contains various pandas DataFrames with different structures
    # g contains mixed data types
    # missing data is also included
    'a': [
        'aa', 'bb', 'cc', 'dd',
        'aa', 'bb', 'cc', 'dd',
        'aa', 'bb', 'cc', 'dd'
    ],
    'b': [
        np.random.random((10, 5)),
        np.nan,
```

(continues on next page)

(continued from previous page)

```

np.random.random((10, 5)),
np.random.random((10, 5)),
np.random.random((10, 5)),
np.random.random((10, 5)),
np.random.random((10, 5)),
np.random.random((10, 5)),
np.random.random((10, 5)),
np.random.random((10, 5)),
np.random.random((10, 5)),
np.random.random((10, 5))
],
'c': [
    {'dicta':[1,2,3], 'dictb':3, 'dictc':{'key1':1, 'key2':2}},
    {'dicta':[52,3], 'dictb':[3,4], 'dictc':{'key4':1, 'key2':2}},
    {'dicta':[12,67], 'dictb':(4,5), 'dictc':{'key3':1, 'key2':77}},
    {'dicta':[1,23], 'dictb':3, 'dictc':{'key1':55, 'key2':33}},
    {'dicta':123, 'dictb':'words', 'dictc':{'key1':4, 'key2':2}},
    {'dicta':[1,2,3], 'dictb':3, 'dictc':{'key1':1, 'key2':2}},
    {'dicta':[52,3], 'dictb':[3,4], 'dictc':{'key4':1, 'key2':2}},
    {'dicta':[12,67], 'dictb':(4,5), 'dictc':{'key3':1, 'key2':77}},
    {'dicta':[1,23], 'dictb':3, 'dictc':{'key1':55, 'key2':33}},
    {'dicta':123, 'dictb':'words', 'dictc':{'key1':4, 'key2':2}},
    {'dicta':[1,2,3], 'dictb':3, 'dictc':{'key1':1, 'key2':2}},
    {'dicta':[52,3], 'dictb':[3,4], 'dictc':{'key4':1, 'key2':2}},
],
'd': [
    np.random.random((16, 5)),
    np.random.random((18, 5)),
    np.random.random((19, 5)),
    np.random.random((11, 5)),
    np.random.random((12, 5)),
    np.random.random((14, 5)),
    np.random.random((17, 5)),
    np.random.random((110, 5)),
    None,
    np.random.random((2, 5)),
    np.random.random((4, 5)),
    np.random.random((7, 5))
],
'e': [
    pd.DataFrame(
        {'c1':[1,2,3], 'c2':[1,2,3]},
        index=pd.MultiIndex.from_arrays(
            [['a', 'b', 'c'], ['a', 'b', 'c']],
            names=['a', 'b']
        )
    ),
    pd.DataFrame(
        {'c1':[1,2,3,4], 'c2':[1,2,3,4]},
        index=pd.MultiIndex.from_arrays(
            [['a', 'b', 'c', 'd'], ['a', 'b', 'c', 'd']],
            names=['a', 'b']
        )
    ),
    pd.DataFrame(
        {'c1':[3,4,3], 'c2':[3,5,3]},
        index=pd.MultiIndex.from_arrays(

```

(continues on next page)

(continued from previous page)

```

        [['a', 'b', 'c'], ['a', 'b', 'c']],
        names=['a', 'b']
    )
),
np.nan,
pd.DataFrame(
    {'c1':[1,2,3], 'c2':[1,2,3]},
    index=pd.MultiIndex.from_arrays(
        [['a', 'b', 'c'], ['a', 'b', 'c']],
        names=['a', 'b']
    )
),
pd.DataFrame(
    {'c1':[1,2,3,4], 'c2':[1,2,3,4]},
    index=pd.MultiIndex.from_arrays(
        [['a', 'b', 'c', 'd'], ['a', 'b', 'c', 'd']],
        names=['a', 'b']
    )
),
pd.DataFrame(
    {'c1':[3,4,3], 'c2':[3,5,3]},
    index=pd.MultiIndex.from_arrays(
        [['a', 'b', 'c'], ['a', 'b', 'c']],
        names=['a', 'b']
    )
),
np.nan,
pd.DataFrame(
    {'c1':[1,2,3], 'c2':[1,2,3]},
    index=pd.MultiIndex.from_arrays(
        [['a', 'b', 'c'], ['a', 'b', 'c']],
        names=['a', 'b']
    )
),
pd.DataFrame(
    {'c1':[1,2,3,4], 'c2':[1,2,3,4], 'c3':[1,2,3,4]},
    index=pd.MultiIndex.from_arrays(
        [['a', 'b', 'c', 'd'], ['a', 'b', 'c', 'd']],
        names=['a', 'b']
    )
),
pd.DataFrame(
    {'c1':[3,4,3], 'c2':[3,5,3]},
    index=pd.MultiIndex.from_arrays(
        [['a', 'b', 'c'], ['a', 'b', 'c']],
        names=['a', 'b']
    )
),
np.nan,
],
'f': [
    pd.DataFrame(
        {'f1':[1,2,3], 'hh2':[1,2,3]},
        index=pd.MultiIndex.from_arrays(
            [['a', 'b', 'c'], ['a', 'b', 'c'], ['f', 'f', 'f']],
            names=['f', 'b', 'e']
        )
    )
]

```

(continues on next page)

(continued from previous page)

```

    ),
    pd.DataFrame(
        {'hh1':[1,2,3,4], 'qq2':[1,2,3,4]},
        index=pd.MultiIndex.from_arrays(
            [['a', 'b', 'c', 'd'], ['a', 'b', 'c', 'd']],
            names=['a', 'b']
        )
    ),
    pd.DataFrame(
        {'q1':[3,4,3], 'qq2':[3,5,3], 'c3':[1,2,3], 'c4':[1,2,3]},
        index=pd.MultiIndex.from_arrays(
            [['a', 'b', 'c'], ['a', 'b', 'c'], ['t', 't', 't']],
            names=['y', 'll', 'tt']
        )
    ),
    np.nan,
    pd.DataFrame(
        {'qq1':[1,2,3], 'rr2':[1,2,3]},
        index=pd.MultiIndex.from_arrays(
            [['a', 'b', 'c'], ['a', 'b', 'c']],
            names=['a', 'b']
        )
    ),
    pd.DataFrame(
        [[1,2,3,4], [1,2,3,4]],
        columns=pd.MultiIndex.from_arrays(
            [['a', 'b', 'c', 'd'], ['a', 'b', 'c', 'd']],
            names=['rr', 'b']
        ),
        index=pd.MultiIndex.from_arrays(
            [(1,2), (2,3)], ['a', 'b']],
            names=['a', 'b']
        )
    ),
    pd.DataFrame(
        {'cpp1':[3,4,3], 'c2':[3,5,3]},
        index=pd.MultiIndex.from_arrays(
            [['a', 'b', 'c'], ['a', 'b', 'c']],
            names=['a', 'rr']
        )
    ),
    np.nan,
    pd.DataFrame(
        {'sr1':[1,2,3,4], 'c2':[1,2,3,4]},
        index=pd.MultiIndex.from_arrays(
            [['a', 'b', 'c', 'd'], ['a', 'b', 'c', 'd']],
            names=['a', 'b']
        )
    ),
    pd.DataFrame(
        {'cpp1':[3,4,3], 'c2':[3,5,3]},
        index=pd.MultiIndex.from_arrays(
            [['a', 'b', 'c'], ['a', 'b', 'c']],
            names=['a', 'b']
        )
    ),
    pd.DataFrame(

```

(continues on next page)

(continued from previous page)

```

        {'c1':[3,4,3], 'c2':[3,5,3]},
        index=pd.MultiIndex.from_arrays(
            [['a', 'b', 'c'], ['a', 'b', 'c']],
            names=['mm', 'b']
        )
    ),
    np.nan,
],
'g': [
    'a', 'b', {'ff':'gg'}, {'a', 'b', 'c'},
    ('r',), pd.Series({'a':'b'}), 'a', 'b',
    1, 2, 3, 4
]
})
df

```

```

[2]:      a                                     b \
0  aa  [[0.9657556404566287, 0.6105982811179597, 0.71...
1  bb                                     NaN
2  cc  [[0.2978295126291556, 0.2876008891935764, 0.85...
3  dd  [[0.4963928696663038, 0.7239167468580807, 0.98...
4  aa  [[0.3031081932962224, 0.6450296792517578, 0.32...
5  bb  [[0.5167332905196971, 0.7313676911394652, 0.58...
6  cc  [[0.708744333480539, 0.8321196509452965, 0.132...
7  dd  [[0.38385169069805636, 0.4351602576907261, 0.2...
8  aa  [[0.8448759939899335, 0.3957149482929728, 0.70...
9  bb  [[0.19421217606406949, 0.7404434981173305, 0.5...
10 cc  [[0.5360890315477351, 0.11323478357643058, 0.7...
11 dd  [[0.22093473502397243, 0.024389277765600403, 0...

                                     c \
0  {'dicta': [1, 2, 3], 'dictb': 3, 'dictc': {'ke...
1  {'dicta': [52, 3], 'dictb': [3, 4], 'dictc': {...
2  {'dicta': [12, 67], 'dictb': (4, 5), 'dictc': ...
3  {'dicta': [1, 23], 'dictb': 3, 'dictc': {'key1...
4  {'dicta': 123, 'dictb': 'words', 'dictc': {'ke...
5  {'dicta': [1, 2, 3], 'dictb': 3, 'dictc': {'ke...
6  {'dicta': [52, 3], 'dictb': [3, 4], 'dictc': {...
7  {'dicta': [12, 67], 'dictb': (4, 5), 'dictc': ...
8  {'dicta': [1, 23], 'dictb': 3, 'dictc': {'key1...
9  {'dicta': 123, 'dictb': 'words', 'dictc': {'ke...
10 {'dicta': [1, 2, 3], 'dictb': 3, 'dictc': {'ke...
11 {'dicta': [52, 3], 'dictb': [3, 4], 'dictc': {...

                                     d \
0  [[0.8332176695108217, 0.789958044060405, 0.294...
1  [[0.3132246893884286, 0.1335576065684959, 0.42...
2  [[0.06184828264219733, 0.18545094706008847, 0...
3  [[0.8993257213551965, 0.2031570590614975, 0.66...
4  [[0.5945125943356216, 0.15692780064527623, 0.0...
5  [[0.6826944216406796, 0.06614703871524041, 0.4...
6  [[0.6154355882964945, 0.5022137513822291, 0.64...
7  [[0.252611789308698, 0.04665515687154631, 0.04...
8                                     None
9  [[0.7887921954495609, 0.07707935123200094, 0.5...
10 [[0.09228121163345293, 0.02214142758664739, 0...

```

(continues on next page)

(continued from previous page)

```

11  [[0.5195772094908402, 0.2494521739025667, 0.39...

                                e  \

0      c1  c2
a b
a a  1  1
b b  2  ...
1      c1  c2
a b
a a  1  1
b b  2  ...
2      c1  c2
a b
a a  3  3
b b  4  ...
3                                NaN
4      c1  c2
a b
a a  1  1
b b  2  ...
5      c1  c2
a b
a a  1  1
b b  2  ...
6      c1  c2
a b
a a  3  3
b b  4  ...
7                                NaN
8      c1  c2
a b
a a  1  1
b b  2  ...
9      c1  c2  c3
a b
a a  1  1  ...
10     c1  c2
a b
a a  3  3
b b  4  ...
11                                NaN

                                f                                g

0      f1  hh2
f b e
a a f  1  1
b...                                a
1      hh1  qq2
a b
a a  1  1
b b ...                                b
2      q1  qq2  c3  c4
y ll tt                                {'ff': 'gg'}
3                                NaN                                {a, c, b}
4      qq1  rr2
a b
a a  1  1

```

(continues on next page)

(continued from previous page)

```

b b ... (r,)
5 rr a b c d
b a b c d
a ... a b
dtype: object
6 cpp1 c2
a rr
a a 3 3
b... a
7 NaN b
8 sr1 c2
a b
a a 1 1
b b ... 1
9 cpp1 c2
a b
a a 3 3
b b ... 2
10 c1 c2
mm b
a a 3 3
b b ... 3
11 NaN 4

```

So this dataframe is quite daunting, I like to call it a *puffry* table.

Exploding the data out that are in puffry tables

Now with `puffry_to_long` you can easily unravel this dataframe. Since this dataframe is weirdly constructed `puffry_to_long` may take a while:

```
[3]: long_df = pb.puffry_to_long(df)
long_df.head()
```

```
[3]:
```

	index_level0	a	b_level0	b_level1	b	c_level0	c_level1	c	\
0	0	aa	0.0	0.0	0.965756	dicta	0	1	
1	0	aa	0.0	0.0	0.965756	dicta	0	1	
2	0	aa	0.0	0.0	0.965756	dicta	0	1	
3	0	aa	0.0	0.0	0.965756	dicta	0	1	
4	0	aa	0.0	0.0	0.965756	dicta	0	1	

	d_level0	d_level1	d	e_level0_a	e_level0_b	e_level0_2	e	\
0	0.0	0.0	0.833218	a	a	c1	1.0	
1	0.0	0.0	0.833218	a	a	c1	1.0	
2	0.0	0.0	0.833218	a	a	c1	1.0	
3	0.0	0.0	0.833218	a	a	c1	1.0	
4	0.0	0.0	0.833218	a	a	c1	1.0	

	f_level0_0	f_level0_1	f	g_level0	g
0	0.0	b	a	NaN	a
1	0.0	e	f	NaN	a
2	0.0	f	a	NaN	a
3	0.0	f1	1	NaN	a
4	0.0	hh2	1	NaN	a

Now we have a dataframe with only hashable elements. `puffry_to_long` iteratively exploded all cells and treated each column individually. For example, if a cell contains a numpy array that is two-dimensional then the column

will be exploded twice and two new columns will be added that are called `*[COLUMN_NAME]_level0*` and `*[COLUMN_NAME]_level1*`. For our column `b`, we get two new columns called `b_level0` and `b_level1`. These levels will contain the index corresponding to the data point in the long-format of column `b`. Let's try `puffy_to_long` again but just on column `b`.

Exploding numpy.array-containing columns

```
[4]: long_df = pb.puffy_to_long(df, 'b')
long_df.head()
```

```
[4]:   index_level0  b_level0  b_level1      b
0              0          0          0  0.965756
1              0          0          1  0.610598
2              0          0          2  0.710187
3              0          0          3  0.851220
4              0          0          4  0.264982
```

`index_level0` is the previous index from our dataframe. If this were a `pandas.MultiIndex`, we would have multiple columns instead of just one that corresponds to the old index of the dataframe.

Now, we could take this normal dataframe objects and perform various operations that we would normally want to perform, e.g.:

```
[5]: long_df.groupby('b_level0')['b'].mean()
```

```
[5]: b_level0
0      0.489353
1      0.538500
2      0.502751
3      0.490095
4      0.505218
5      0.501370
6      0.474048
7      0.573405
8      0.491318
9      0.516571
Name: b, dtype: float64
```

Let's say we want to *explode* both column `b` and column `d` that both contain numpy arrays. Imagine that we want to align the axis 1 of the data in `b` and `d`, and call this axes `aligned_axis`, we can do this with `puffy_to_long` keyword arguments:

```
[6]: long_df = pb.puffy_to_long(df, 'b', 'd', aligned_axis={'b':1, 'd':1})
long_df.head()
```

```
[6]:   index_level0  b_level0  aligned_axis      b  d_level0      d
0              0          0.0            0  0.965756      0.0  0.833218
1              0          0.0            0  0.965756      1.0  0.060445
2              0          0.0            0  0.965756      2.0  0.620360
3              0          0.0            0  0.965756      3.0  0.610501
4              0          0.0            0  0.965756      4.0  0.671548
```

So now, axis 1 of `b` and `d` column data are aligned and those indices are defined in the `aligned_axis` column. Since both columns contain missing cells, some `index_level0` values have missing `b` and `b_level0` columns or missing `d_level0` and `d` columns. You can view these easily using standard pandas functionality:

```
[7]: long_df.loc[long_df['b_level0'].isnull()].head()
```

```
[7]:
```

	index_level0	b_level0	aligned_axis	b	d_level0	d
10650	1	NaN	0	NaN	0.0	0.313225
10651	1	NaN	0	NaN	1.0	0.630630
10652	1	NaN	0	NaN	2.0	0.074653
10653	1	NaN	0	NaN	3.0	0.290296
10654	1	NaN	0	NaN	4.0	0.563310

```
[8]: long_df.loc[long_df['d_level0'].isnull()].head()
```

```
[8]:
```

	index_level0	b_level0	aligned_axis	b	d_level0	d
9950	8	0.0	0	0.844876	NaN	NaN
9951	8	1.0	0	0.905452	NaN	NaN
9952	8	2.0	0	0.567938	NaN	NaN
9953	8	3.0	0	0.043224	NaN	NaN
9954	8	4.0	0	0.736017	NaN	NaN

Exploding pandas.DataFrame-containing columns

Let's take a look at how `pandas.DataFrame` objects are handled within `puffy_to_long` by taking a look at column `e`:

```
[9]: long_df = pb.puffy_to_long(df, 'e')
long_df.head()
```

```
[9]:
```

	index_level0	e_level0_a	e_level0_b	e_level0_2	e
0	0	a	a	c1	1.0
1	0	a	a	c2	1.0
2	0	b	b	c1	2.0
3	0	b	b	c2	2.0
4	0	c	c	c1	3.0

`pandas.DataFrame` objects are handled within one *explosion* iteration, unless the cell within that dataframe are non-hashable. This is why all new columns contain `level0`. The first two new columns `e_level0_a` and `e_level0_b` correspond to the `pandas.MultiIndex` index defined in all dataframes within this column. `e_level0_2` corresponds to all the columns names of the dataframe. `e` only contains the data within each cell of each dataframe.

Let's say we don't want to unravel our columns in this way but instead just concatenate them all together. We can use the `expand_cols` argument for this, which expects a list of column names that contain only `pandas.DataFrame` or `pandas.Series` objects:

```
[10]: long_df = pb.puffy_to_long(df, 'e', expand_cols=['e'])
long_df.head()
```

```
[10]:
```

	index_level0	a	b	e_c1	e_c2	e_c3
0	0	a	a	1	1	NaN
1	0	b	b	2	2	NaN
2	0	c	c	3	3	NaN
3	1	a	a	1	1	NaN
4	1	b	b	2	2	NaN

Here we preserved the columns of each dataframe in each cell and simply concatenated the dataframes together with the `index_level0` information preserved. What if we use this method while also *exploding* column `a`, since this has the same column name as the column in the dataframes within column `e`:

```
[11]: long_df = pb.puffy_to_long(df, 'a', 'e', expand_cols=['e'])
long_df.head()
```



```
[11]:
```

	index_level0	a	a_e	b	e_c1	e_c2	e_c3
0	0	aa	a	a	1.0	1.0	NaN
1	0	aa	b	b	2.0	2.0	NaN
2	0	aa	c	c	3.0	3.0	NaN
3	1	bb	a	a	1.0	1.0	NaN
4	1	bb	b	b	2.0	2.0	NaN

Since the column a already existed, the column a within each dataframe within column e was renamed to a_e.

Of course, similarly, this is handled with column b:

```
[12]: long_df = pb.puffly_to_long(df, 'b', 'e', expand_cols=['e'])
long_df.head()
```

```
[12]:
```

	index_level0	b_level0	b_level1	b	a	b_e	e_c1	e_c2	e_c3
0	0	0.0	0.0	0.965756	a	a	1.0	1.0	NaN
1	0	0.0	0.0	0.965756	b	b	2.0	2.0	NaN
2	0	0.0	0.0	0.965756	c	c	3.0	3.0	NaN
3	0	0.0	1.0	0.610598	a	a	1.0	1.0	NaN
4	0	0.0	1.0	0.610598	b	b	2.0	2.0	NaN

Less structured dataframe-containing columns will result in more complex long-format dataframes:

```
[13]: long_df = pb.puffly_to_long(df, 'f')
long_df.head()
```

```
[13]:
```

	index_level0	f_level0_0	f_level0_1	f
0	0	0	b	a
1	0	0	e	f
2	0	0	f	a
3	0	0	f1	1
4	0	0	hh2	1

```
[14]: long_df = pb.puffly_to_long(df, 'f', expand_cols=['f'])
long_df.head()
```

```
[14]:
```

	index_level0	level_1	f_f	f_b	f_e	f_f1	f_hh2	f_a	f_hh1	f_qq2	...	\
0	0	0	a	a	f	1.0	1.0	NaN	NaN	NaN	...	
1	0	1	b	b	f	2.0	2.0	NaN	NaN	NaN	...	
2	0	2	c	c	f	3.0	3.0	NaN	NaN	NaN	...	
3	1	0	NaN	a	NaN	NaN	NaN	a	1.0	1.0	...	
4	1	1	NaN	b	NaN	NaN	NaN	b	2.0	2.0	...	

	f_('a', 'a')	f_('b', 'b')	f_('c', 'c')	f_('d', 'd')	f_rr	f_cpp1	f_c2	\
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

	f_sr1	f_mm	f_c1
0	NaN	NaN	NaN
1	NaN	NaN	NaN
2	NaN	NaN	NaN
3	NaN	NaN	NaN
4	NaN	NaN	NaN

[5 rows x 28 columns]

Exploding dictionaries

The column `c` contains dictionaries with various data types that are in them. The `puffy_to_long` algorithm iteratively *explodes* all objects within the dictionaries:

```
[15]: long_df = pb.puffy_to_long(df, 'c')
      long_df.head()
```

```
[15]:   index_level0  c_level0  c_level1  c
0             0      dicta         0  1
1             0      dicta         1  2
2             0      dicta         2  3
3             0      dictb        NaN  3
4             0      dictc        key1  1
```

Since some values within dictionaries can be further exploded, while others cannot some levels/axes contain NaNs when the *explosion* iteration for that data type stopped (for a specific row).

API REFERENCE

<code>puffy_to_long(table, *cols, **kwargs)</code>	Transform the “ <i>puffy</i> ” table into a <i>long-format</i> <code>DataFrame</code> .
<code>FrameEngine(table[, datacols, indexcols, ...])</code>	Class to handle and transform a <code>pandas.DataFrame</code> object.
<code>CallableContainer(default_callable)</code>	Container of callables, that accept one argument.

2.1 puffbird.puffy_to_long

`puffbird.puffy_to_long(table, *cols, **kwargs)`
 Transform the “*puffy*” table into a *long-format* `DataFrame`.

Parameters

- table** [`DataFrame`] A table with “*puffy*” columns.
- cols** [str] A selection of “*data columns*” to create the long dataframe with. If not given, the algorithm will use all “*data columns*”.
- iterable** [callable or dict of callables, optional] This function is called on each cell for each “*data column*” to create a new `Series` object. If the “*data columns*” contains `dict`, `list`, `int`, `float`, `array`, `recarray`, `DataFrame`, or `Series` object types than the default iterable will handle these appropriately. When passing a dictionary of iterables, the keys should correspond to values in `datacols` (i.e. the “*data columns*” of the *table*). In this case, each column can have a custom iterable used. If a column’s iterable is not specified the default iterable is used.
- max_depth** [int or dict of ints, optional] Maximum depth of expanding each cell, before the algorithm stops for each “*data column*”. If we set the `max_depth` to 3, for example, a “*data column*” consisting of 4-D `array` objects will result in a `DataFrame` where the “*data column*” cells contain 1-D `array` objects. If the arrays were 3-D, it will result in a long dataframe with scalars in each cell. Defaults to 3.
- dropna** [bool, optional] Drop rows in *long-format* `DataFrame`, where **all** “*data columns*” are NaNs.
- cond** [callable or dict of callables, optional] This function should return *True* or *False* and accept a `Series` object as an argument. If *True*, the algorithm will stop “*exploding*” a “*data column*”. The default *cond* argument suffices for all non-hashable types, such as `list` or `array` objects. If you want to “*explode*” hashable types such as `tuple` objects, a custom *cond* callable has to be defined. However, it is recommended that hashable types are first converted into non-hashable types using a custom conversion function and the `col_apply` method.

expand_cols [list-like, optional] Specify a list of “data columns” to apply the `expand_col` method instead of “exploding” the column in the table. If all cells within a “data column” contains similarly constructed `DataFrame` or `Series` object types, the `expand_col` method can be used instead of “exploding” the “data column”. Default to None.

shared_axes [dict, optional] Specify if two or more “data columns” share axes (i.e. “explosion” iterations). The keyword will correspond to what the column will be called in the long dataframe. Each argument is a dictionary where the keys correspond to the names of the “data columns”, which share an axis, and the value correspond to the depth/axis is shared for each “data column”. `shared_axis` argument is usually defined for “data columns” that contain `array` objects. For example, one “data column” may consists of one-dimensional timestamp arrays and another “data column” may consist of two-dimensional timeseries arrays where the first axis of the latter is shared with the zeroth axis of the former.

Returns

DataFrame A long-format `DataFrame`.

See also:

`FrameEngine.to_long`

`FrameEngine.expand_col`

Notes

If you find yourself writing custom *iterable* and *cond* arguments and believe these may be of general use, please open an [issue](#) or start a pull request.

Examples

```
>>> import pandas as pd
>>> import puffbird as pb
>>> df = pd.DataFrame({
...     'a': [[1,2,3], [4,5,6,7], [3,4,5]],
...     'b': [{'c':['asdf'], 'd':['ret']}, {'d':['r']}, {'c':['ff']}],
... })
>>> df
```

	a	b
0	[1, 2, 3]	{'c': ['asdf'], 'd': ['ret']}
1	[4, 5, 6, 7]	{'d': ['r']}
2	[3, 4, 5]	{'c': ['ff']}

Now we can use the `puffy_to_long` function to create a long-format `DataFrame`:

```
>>> pb.puffy_to_long(df)
```

	index_level0	a_level0	a	b_level0	b_level1	b
0	0	0	1.0	c	0	asdf
1	0	0	1.0	d	0	ret
2	0	1	2.0	c	0	asdf
3	0	1	2.0	d	0	ret
4	0	2	3.0	c	0	asdf
5	0	2	3.0	d	0	ret
6	1	0	4.0	d	0	r
7	1	1	5.0	d	0	r
8	1	2	6.0	d	0	r

(continues on next page)

(continued from previous page)

9	1	3	7.0	d	0	r
10	2	0	3.0	c	0	ff
11	2	1	4.0	c	0	ff
12	2	2	5.0	c	0	ff

2.2 puffbird.FrameEngine

```
class puffbird.FrameEngine(table, datacols=None, indexcols=None, inplace=False, handle_column_types=True, enforce_identifier_string=False, fast_path=False)
```

Class to handle and transform a `pandas.DataFrame` object.

Parameters

table [`DataFrame`] A table with singular `Index` columns, where each column corresponds to a specific data type. `MultiIndex` columns will be made singular with the `to_flat_index` method. It is recommended that all columns and index names are identifier string types. Individual cells within `datacols` columns may have arbitrary objects in them, but cells within `indexcols` columns must be hashable.

datacols [list-like, optional] The columns in `table` that are considered “data”. For example, columns where each cell is a `numpy.array` object. If `None`, all columns are considered `datacols` columns, unless `indexcols` is specified. Defaults to `None`.

indexcols [list-like, optional] The columns in `table` that are immutable or hashable types, e.g. strings or integers. These may correspond to “metadata” that describe or specify the `datacols` columns. If `None`, only the index of the `table`, which may be `MultiIndex`, are considered `indexcols` columns. If `datacols` is specified and `indexcols` is `None`, then the remaining columns are also added to the index of `table`. Defaults to `None`.

inplace [bool, optional] If possible do not copy the `table` object. Defaults to `False`.

handle_column_types [bool, optional] If `True`, converts not string column types to strings. Defaults to `True`.

enforce_identifier_string [bool, optional] If `True`, try to convert all types to identifier string types and check if all columns are identifier string types. Enforcement only works if column types are `str`, `Number`, or `tuple` object types. Throw an error if enforcement does not work. Defaults to `False`.

Notes

A `table` has singular `Index` columns, where each column corresponds to a specific data type. These types of tables are often fetched from databases that use data models such as `datajoint`. The `table` often needs to be transformed, so that various computations such as `groupby` can be performed or the data can be plotted easily with packages such as `seaborn`. In the `table`, the columns and the index names are considered together and divided into `datacols` and `indexcols`. “Data columns” are usually columns that contain Python objects that are iterable and need to be “exploded” in order to convert these columns into numeric or other immutable data types. This is why I call these types of tables “puffy” dataframes. “Index columns” usually contain other information, often considered “metadata”, that uniquely identify each row. Each row for a specific column is considered to have the same data type and can thus be “exploded” the same way. Missing data (NaNs) are allowed.

Examples

```
>>> import pandas as pd
>>> import puffbird as pb
>>> df = pd.DataFrame({
...     'a': [[1,2,3], [4,5,6,7], [3,4,5]],
...     'b': [{'c':['asdf'], 'd':['ret']}, {'d':['r']}, {'c':['ff']}],
... })
>>> df
      a      b
0  [1, 2, 3]  {'c': ['asdf'], 'd': ['ret']}
1  [4, 5, 6, 7]  {'d': ['r']}
2  [3, 4, 5]  {'c': ['ff']}
```

The `FrameEngine` instance has various methods that allow for quick manipulation of this “puffy” dataframe. For example, we can create a long dataframe using the `to_long()` method:

```
>>> engine.to_long()
  index_col_0  b_level0  b_level1  b  a_level0  a
0           0         c         0  asdf         0  1.0
1           0         c         0  asdf         1  2.0
2           0         c         0  asdf         2  3.0
3           0         d         0  ret          0  1.0
4           0         d         0  ret          1  2.0
5           0         d         0  ret          2  3.0
6           1         d         0   r           0  4.0
7           1         d         0   r           1  5.0
8           1         d         0   r           2  6.0
9           1         d         0   r           3  7.0
10          2         c         0  ff           0  3.0
11          2         c         0  ff           1  4.0
12          2         c         0  ff           2  5.0
```

Attributes

<code>cols</code>	Tuple of “data columns” and “index columns” in the table.
<code>cols_rename</code>	Mapping of renamed “data columns” and “index columns” in table.
<code>datacols</code>	Tuple of the “data columns” in the table.
<code>datacols_rename</code>	Mapping of renamed “data columns” in table.
<code>indexcols</code>	Tuple of the “index columns” in the table.
<code>indexcols_rename</code>	Mapping of renamed “index columns” in table.
<code>table</code>	<code>DataFrame</code> passed during initialization.

2.2.1 puffbird.FrameEngine.cols

property `FrameEngine.cols`

Tuple of “*data columns*” and “*index columns*” in the *table*.

2.2.2 puffbird.FrameEngine.cols_rename

property `FrameEngine.cols_rename`

Mapping of renamed “*data columns*” and “*index columns*” in *table*.

2.2.3 puffbird.FrameEngine.datacols

property `FrameEngine.datacols`

Tuple of the “*data columns*” in the *table*.

See also:

`pandas.DataFrame.columns`

2.2.4 puffbird.FrameEngine.datacols_rename

property `FrameEngine.datacols_rename`

Mapping of renamed “*data columns*” in *table*.

2.2.5 puffbird.FrameEngine.indexcols

property `FrameEngine.indexcols`

Tuple of the “*index columns*” in the *table*.

See also:

`pandas.MultiIndex.names`

2.2.6 puffbird.FrameEngine.indexcols_rename

property `FrameEngine.indexcols_rename`

Mapping of renamed “*index columns*” in *table*.

2.2.7 puffbird.FrameEngine.table

property `FrameEngine.table`

`DataFrame` passed during initialization.

Methods

<code>apply(func, new_col_name, *args[, ...])</code>	Apply a function to each row in the <i>table</i> .
<code>col_apply(func, col[, new_col_name, ...])</code>	Apply a function to a specific column in each row in the <i>table</i> .
<code>drop(*cols[, skip, skip_index, skip_data])</code>	Drop columns in place.
<code>expand_col(col[, reset_index, dropna, ...])</code>	Expand a column that contain <code>DataFrame</code> or <code>Series</code> object types to create a single <i>long-format</i> <code>DataFrame</code> .
<code>multid_pivot([values])</code>	Pivot the <i>table</i> to create a multidimensional <code>xarray.DataArray</code> or <code>xarray.DataSet</code> object.
<code>rename(**rename_kws)</code>	Rename columns in place.
<code>to_long(*cols[, iterable, max_depth, ...])</code>	Transform the “ <i>puffy</i> ” table into a <i>long-format</i> <code>DataFrame</code> .
<code>to_puffy(*indexcols[, keep_missing_idcs, ...])</code>	Make the <i>table</i> “ <i>puffier</i> ” by aggregating across unique sets of “ <i>index columns</i> ”.

2.2.8 puffbird.FrameEngine.apply

`FrameEngine.apply(func, new_col_name, *args, assign_to_index=False, map_kws=None, **kwargs)`

Apply a function to each row in the *table*.

Parameters

func [callable] Function to apply. The function cannot return a `Series` object.

new_col_name [str] Name of computed new column. If None, *new_col_name* will be “*apply_result*”.

args [tuple] Arguments passed to function. Each argument should be an “*index column*” or “*data column*” in the *table*. Thus, the argument will correspond to the cell value for each row.

assign_to_index [bool, optional] Assign new column as “*index column*”, instead of as “*data column*”..

map_kws [dict] Same as args just as keyword arguments.

kwargs: dict Keyword arguments passed to function as is.

Returns

self

2.2.9 puffbird.FrameEngine.col_apply

`FrameEngine.col_apply(func, col, new_col_name=None, assign_to_index=None, **kwargs)`

Apply a function to a specific column in each row in the *table*.

Parameters

func [callable] Function to apply. The function cannot return a `Series` object.

col [str] Name of “*data column*”.

new_col_name [str, optional] Name of computed new column. If `None`, this will be set to the name of the column; i.e. the name of the column will be overwritten. Defaults to `None`.

assign_to_index [bool, optional] Assign new column as “*index column*”, instead of as “*data column*”..

kwargs [dict] Keyword Arguments passed each function call.

Returns

`self`

2.2.10 puffbird.FrameEngine.drop

`FrameEngine.drop(*cols, skip=False, skip_index=False, skip_data=False)`

Drop columns in place.

Parameters

cols [str] Columns to drop.

skip [bool] If `True`, skip values in *cols* that do not match with any columns. Defaults to `False`.

skip_index [bool] If `True`, skip dropping “*index columns*”. Defaults to `False`.

skip_data [bool] If `True`, skip dropping “*data columns*”. Defaults to `False`.

Returns

`self`

See also:

`pandas.DataFrame.drop`

2.2.11 puffbird.FrameEngine.expand_col

`FrameEngine.expand_col(col, reset_index=True, dropna=True, handle_diff=True)`

Expand a column that contain `DataFrame` or `Series` object types to create a single *long-format DataFrame*.

Parameters

col [str] The “*data column*” to expand.

reset_index [bool, optional] Whether to reset the index of the new *long-format DataFrame*. Defaults to `True`.

dropna [bool, optional] Whether to drop NaNs in the “*data column*”. If `False` and NaNs exist, this will currently result in an error. Defaults to `True`.

handle_diff [bool, optional] Handle indices across column cells, if they cannot be concatenated, instead of throwing an error. Defaults to True.

Returns

DataFrame or Series A long-format DataFrame or Series.

See also:

FrameEngine.to_long

puffry_to_long

2.2.12 puffbird.FrameEngine.multid_pivot

FrameEngine.**multid_pivot** (*values=None, *dims*)

Pivot the *table* to create a multidimensional xarray.DataArray or xarray.DataSet object.

Warning: This method has not yet been implemented. It will be defined in future releases.

2.2.13 puffbird.FrameEngine.rename

FrameEngine.**rename** (***rename_kws*)

Rename columns in place.

Parameters

rename_kws [dict] Mapping of old column names to new column names.

Returns

self

See also:

pandas.DataFrame.rename

2.2.14 puffbird.FrameEngine.to_long

FrameEngine.**to_long** (**cols, iterable=CallableContainer(iter), max_depth=3, dropna=True, reindex=False, cond=is_hashable(series), expand_cols=None, **shared_axes*)

Transform the “puffry” table into a long-format DataFrame.

Parameters

cols [str] A selection of “data columns” to create the long dataframe with. If not given, the algorithm will use all “data columns”.

iterable [callable or dict of callables, optional] This function is called on each cell for each “data column” to create a new Series object. If the “data columns” contains dict, list, int, float, array, recarray, DataFrame, or Series object types than the default iterable will handle these appropriately. When passing a dictionary of iterables, the keys should correspond to values in *datacols* (i.e. the “data columns” of the *table*). In this case, each column can have a custom iterable used. If a column’s iterable is not specified the default iterable is used.

max_depth [int or dict of ints, optional] Maximum depth of expanding each cell, before the algorithm stops for each “*data column*”. If we set the `max_depth` to 3, for example, a “*data column*” consisting of 4-D `array` objects will result in a `DataFrame` where the “*data column*” cells contain 1-D `array` objects. If the arrays were 3-D, it will result in a long dataframe with scalars in each cell. Defaults to 3.

dropna [bool, optional] Drop rows in *long-format DataFrame*, where **all** “*data columns*” are NaNs.

cond [callable or dict of callables, optional] This function should return *True* or *False* and accept a `Series` object as an argument. If *True*, the algorithm will stop “*exploding*” a “*data column*”. The default `cond` argument suffices for all non-hashable types, such as `list` or `array` objects. If you want to “*explode*” hashable types such as `tuple` objects, a custom `cond` callable has to be defined. However, it is recommended that hashable types are first converted into non-hashable types using a custom conversion function and the `col_apply` method.

expand_cols [list-like, optional] Specify a list of “*data columns*” to apply the `expand_col` method instead of “*exploding*” the column in the table. If all cells within a “*data column*” contains similarly constructed `DataFrame` or `Series` object types, the `expand_col` method can be used instead of “*exploding*” the “*data column*”. Default to `None`.

shared_axes [dict, optional] Specify if two or more “*data columns*” share axes (i.e. “*explosion*” iterations). The keyword will correspond to what the column will be called in the long dataframe. Each argument is a dictionary where the keys correspond to the names of the “*data columns*”, which share an axis, and the value correspond to the depth/axis is shared for each “*data column*”. `shared_axis` argument is usually defined for “*data columns*” that contain `array` objects. For example, one “*data column*” may consists of one-dimensional timestamp arrays and another “*data column*” may consist of two-dimensional timeseries arrays where the first axis of the latter is shared with the zeroth axis of the former.

Returns

`DataFrame` A *long-format DataFrame*.

See also:

[`puffy_to_long`](#)

Notes

If you find yourself writing custom *iterable* and *cond* arguments and believe these may be of general use, please open an [issue](#) or start a pull request.

Examples

```
>>> import pandas as pd
>>> import puffbird as pb
>>> df = pd.DataFrame({
...     'a': [[1,2,3], [4,5,6,7], [3,4,5]],
...     'b': [{'c':['asdf'], 'd':['ret']}, {'d':['r']}, {'c':['ff']}],
... })
>>> df
```

a
b

(continues on next page)

(continued from previous page)

```

0      [1, 2, 3]  {'c': ['asdf'], 'd': ['ret']}
1      [4, 5, 6, 7]  {'d': ['r']}
2      [3, 4, 5]  {'c': ['ff']}
>>> engine = pb.FrameEngine(df)

```

Now we can use the `to_long` method to create a *long-format* `DataFrame`:

```

>>> engine.to_long()
   index_level0  a_level0    a  b_level0  b_level1    b
0              0         0  1.0         c         0  asdf
1              0         0  1.0         d         0   ret
2              0         1  2.0         c         0  asdf
3              0         1  2.0         d         0   ret
4              0         2  3.0         c         0  asdf
5              0         2  3.0         d         0   ret
6              1         0  4.0         d         0     r
7              1         1  5.0         d         0     r
8              1         2  6.0         d         0     r
9              1         3  7.0         d         0     r
10             2         0  3.0         c         0   ff
11             2         1  4.0         c         0   ff
12             2         2  5.0         c         0   ff

```

2.2.15 puffbird.FrameEngine.to_puffry

`FrameEngine.to_puffry(*indexcols, keep_missing_idcs=True, aggfunc=CallableContainer(list), dropna=True)`

Make the *table* “*puffier*” by aggregating across unique sets of “*index columns*”.

Warning: `to_puffry` is currently an experimental method and so it may change significantly in future releases.

Parameters

indexcols [str] Set of “*index columns*” to aggregate over using `groupby`.

keep_missing_idcs [bool, optional] If True, aggregate index columns not in the *indexcols* argument. Defaults to True.

aggfunc [callable or dict of callables, optional] The function used to aggregate “*data columns*” and any missing indices. Defaults to list.

dropna [bool, optional] Drop NaNs in table before aggregating.

Returns

`DataFrame`

See also:

`FrameEngine.to_long`

2.3 puffbird.CallableContainer

class puffbird.CallableContainer (*default_callable*)

Container of callables, that accept one argument.

Parameters

default_callable [callable] Default callable used when argument passed does not correspond to an assigned instance.

Notes

Each callable can be assigned to a specific instances using the [add](#) method. This method accepts a callable and a class or a tuple of class objects. When the container is called it will check if the single argument passed corresponds to an instance of the class objects and if so use the assigned callable with the argument passed.

Methods

<code>__call__(x)</code>	Check type of <i>x</i> and then use the appropriate callable.
<code>add(a_callable, classes)</code>	Add a new callable with allowed classes.

2.3.1 puffbird.CallableContainer.__call__

CallableContainer.**__call__**(*x*)

Check type of *x* and then use the appropriate callable.

2.3.2 puffbird.CallableContainer.add

CallableContainer.**add**(*a_callable, classes*)

Add a new callable with allowed classes.

{{ header }}

DEVELOPMENT

{{ header }}

3.1 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:

3.1.1 Types of Contributions

Report Bugs

Report bugs at <<https://github.com/gucky92/puffbird/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.

Fix Bugs

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

Implement Features

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

Write Documentation

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

Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/gucky92/puffbird/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 :)

3.1.2 Get Started!

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

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

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

3. Install your local copy into a virtualenv or a conda environment. Assuming you have virtualenvwrapper or conda installed (called puffbird), this is how you set up your fork for local development:

```
$ mkvirtualenv puffbird or conda activate puffbird
$ cd puffbird/
$ 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 puffbird tests
$ python setup.py test or pytest
$ 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.

3.1.3 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. Provide a tutorial in the form of an annotated *.ipynb* file and add it to docs/source/user_guide/tutorials.
3. The pull request should work for Python 3.6, 3.7 and 3.8, and for PyPy. Check https://travis-ci.com/gucky92/puffbird/pull_requests and make sure that the tests pass for all supported Python versions.

3.1.4 Tips

To run a subset of tests:

```
$ pytest tests.test_frameengine.py
```

3.1.5 Deploying

TODO {{ header }}

3.2 Style Guide

puffbird follows the [PEP8](#) standard and uses [Flake8](#) to ensure a consistent code format throughout the project.

3.2.1 Patterns

Using `foo.__class__`

puffbird uses ‘`type(foo)`’ instead ‘`foo.__class__`’ as it is making the code more readable. For example:

Good:

```
foo = "bar"
type(foo)
```

Bad:

```
foo = "bar"
foo.__class__
```

3.2.2 String formatting

Concatenated strings

Using f-strings

puffbird uses f-strings formatting instead of ‘%’ and ‘.format()’ string formatters.

The convention of using f-strings on a string that is concatenated over several lines, is to prefix only the lines containing values which need to be interpreted.

For example:

Good:

```
foo = "old_function"
bar = "new_function"

my_warning_message = (
    f"Warning, {foo} is deprecated, "
    "please use the new and way better "
    f"{bar}"
)
```

Bad:

```
foo = "old_function"
bar = "new_function"

my_warning_message = (
    f"Warning, {foo} is deprecated, "
    f"please use the new and way better "
    f"{bar}"
)
```

White spaces

Only put white space at the end of the previous line, so there is no whitespace at the beginning of the concatenated string.

For example:

Good:

```
example_string = (
    "Some long concatenated string, "
    "with good placement of the "
    "whitespaces"
)
```

Bad:

```
example_string = (
    "Some long concatenated string,"
    " with bad placement of the"
    " whitespaces"
)
```

Representation function (aka 'repr()')

puffbird uses 'repr()' instead of '%r' and '!r'.

The use of 'repr()' will only happen when the value is not an obvious string.

For example:

Good:

```
value = str
f"Unknown received value, got: {repr(value)}"
```

Good:

```
value = str
f"Unknown received type, got: '{type(value).__name__}'"
```

3.2.3 Imports (aim for absolute)

In Python 3, absolute imports are recommended. Using absolute imports, doing something like `import string` will import the string module rather than `string.py` in the same directory. As much as possible, you should try to write out absolute imports that show the whole import chain from top-level puffbird.

Explicit relative imports are also supported in Python 3 but it is not recommended to use them. Implicit relative imports should never be used and are removed in Python 3.

For example:

```
# preferred
from puffbird.frame import FrameEngine

# not preferred
from .frame import FrameEngine

# wrong
from frame import FrameEngine
```

{{ header }}

3.3 Authors

3.3.1 Development Lead

- Matthias Christenson - gucky92: <gucky@gucky.eu>

3.3.2 Contributors

None yet. Why not be the first? {{ header }}

RELEASE NOTES

This is the list of changes to puffbird between each release. For full details, see the [commit logs](#).

4.1 Version 0.0.0

{{ header }}

4.1.1 Version 0.0.1 (May 15, 2020)

Preliminary version of puffbird. It contains the main features, which include the `puffy_to_long` function and the `FrameEngine` object.

Contributors

- Matthias Christenson ([gucky92](#))

PYTHON MODULE INDEX

p

puffbird, [1](#)

Symbols

`__call__()` (*puffbird.CallableContainer method*), 25

A

`add()` (*puffbird.CallableContainer method*), 25

`apply()` (*puffbird.FrameEngine method*), 20

C

`CallableContainer` (*class in puffbird*), 25

`col_apply()` (*puffbird.FrameEngine method*), 21

`cols()` (*puffbird.FrameEngine property*), 19

`cols_rename()` (*puffbird.FrameEngine property*), 19

D

`datacols()` (*puffbird.FrameEngine property*), 19

`datacols_rename()` (*puffbird.FrameEngine property*), 19

`drop()` (*puffbird.FrameEngine method*), 21

E

`expand_col()` (*puffbird.FrameEngine method*), 21

F

`FrameEngine` (*class in puffbird*), 17

I

`indexcols()` (*puffbird.FrameEngine property*), 19

`indexcols_rename()` (*puffbird.FrameEngine property*), 19

M

`module`

`puffbird`, 1

`multid_pivot()` (*puffbird.FrameEngine method*), 22

P

`puffbird`

`module`, 1

`puffy_to_long()` (*in module puffbird*), 15

R

`rename()` (*puffbird.FrameEngine method*), 22

T

`table()` (*puffbird.FrameEngine property*), 19

`to_long()` (*puffbird.FrameEngine method*), 22

`to_puffy()` (*puffbird.FrameEngine method*), 24