pyg

Release 0.0.1

Yoav Git

Mar 09, 2021

CONTENTS

1 README 1

2 pyg.base 3
2.1 extensions to dict . . . 3
2.1.1 dictattr . . . 3
2.1.2 ulist . . . 5
2.1.3 Dict . . . 5
2.1.4 dictable . . . 6
2.1.5 perdictable . . . 15
2.1.6 join . . . 16
2.1.7 named_dict . . . 18
2.2 decorators . . . 19
2.2.1 wrapper . . . 19
2.2.2 timer . . . 20
2.2.3 try_value . . . 21
2.2.4 try_back . . . 21
2.2.5 loops . . . 21
2.2.6 loop . . . 22
2.2.7 kwargs_support . . . 22
2.3 graphs & cells . . . 22
2.3.1 cell . . . 22
2.3.2 cell_go . . . 23
2.3.3 cell_item . . . 24
2.3.4 cell_func . . . 24
2.4 encode and decode/save and load . . . 25
2.4.1 encode . . . 25
2.4.2 decode . . . 26
2.4.3 pd_to_parquet . . . 26
2.4.4 pd_read_parquet . . . 27
2.4.5 parquet_encode . . . 27
2.4.6 csv_encode . . . 28
2.4.7 convertors to bytes . . . 28
2.5 dates and calendar . . . 28
2.5.1 dt . . . 28
2.5.2 ymd . . . 30
2.5.3 dt_bump . . . 30
2.5.4 drange . . . 30
2.5.5 date_range . . . 31
2.5.6 Calendar . . . 31
2.5.7 calendar . . . 34

2.5.8 as_time . . . 34
2.5.9 clock . . . 34
2.6 text manipulation . . . 35
2.6.1 lower . . . 35
2.6.2 upper . . . 35
2.6.3 proper . . . 36
2.6.4 capitalize . . . 36
2.6.5 strip . . . 36
2.6.6 split . . . 37
2.6.7 replace . . . 37
2.6.8 common_prefix . . . 38
2.7 files & directory . . . 38
2.7.1 mkdir . . . 38
2.7.2 read_csv . . . 38
2.8 tree manipulation . . . 38
2.8.1 tree_to_items . . . 38
2.8.2 items_to_tree . . . 39
2.8.3 tree_update . . . 40
2.8.4 tree_to_table . . . 41
2.8.5 tree_repr . . . 42
2.9 list functions . . . 43
2.9.1 as_list . . . 43
2.9.2 as_tuple . . . 43
2.9.3 first . . . 44
2.9.4 last . . . 44
2.9.5 unique . . . 44
2.10 Comparing and Sorting . . . 45
2.10.1 cmp . . . 45
2.10.2 Cmp . . . 45
2.10.3 sort . . . 46
2.10.4 eq . . . 46
2.10.5 in . . . 47
2.11 bits and pieces . . . 47
2.11.1 type functions . . . 47
2.11.2 zipper . . . 48
2.11.3 reducer . . . 49
2.11.4 reducing . . . 50
2.11.5 logger and get_logger . . . 50
2.11.6 access functions . . . 50
2.11.7 inspection . . . 52

3 pyg.mongo **57**
3.1 Query generator . . . 57
3.1.1 q and Q . . . 57
3.2 Tables in Mongo . . . 58
3.2.1 mongo_cursor . . . 58
3.2.2 mongo_reader . . . 62
3.2.3 mongo_pk_reader . . . 66
3.2.4 mongo_pk_cursor . . . 67
3.3 encoding docs before saving to mongo . . . 67
3.3.1 parquet_write . . . 67
3.3.2 csv_write . . . 67
3.4 cells in Mongo . . . 67
3.4.1 db_cell . . . 68

3.4.2 periodic_cell 69
3.4.3 get_cell 69
3.4.4 db_save 69
3.4.5 db_load 70
3.4.6 db_ref 70

4 pyg.timeseries **71**
4.1 simple functions 72
4.1.1 diff 72
4.1.2 shift 72
4.1.3 ratio 73
4.1.4 ts_count 73
4.1.5 ts_sum 74
4.1.6 ts_mean 74
4.1.7 ts_rms 75
4.1.8 ts_std 76
4.1.9 ts_skew 76
4.1.10 ts_min 77
4.1.11 ts_max 77
4.1.12 ts_median 77
4.1.13 fnna 77
4.1.14 v2na/na2v 77
4.1.15 ffill/bfill 78
4.1.16 nona 79
4.2 expanding window functions 79
4.2.1 expanding_mean 79
4.2.2 expanding_rms 80
4.2.3 expanding_std 81
4.2.4 expanding_sum 83
4.2.5 expanding_skew 84
4.2.6 expanding_min 85
4.2.7 expanding_max 86
4.2.8 expanding_median 87
4.2.9 expanding_rank 88
4.2.10 cumsum 88
4.2.11 cumprod 89
4.3 rolling window functions 90
4.3.1 rolling_mean 90
4.3.2 rolling_rms 91
4.3.3 rolling_std 92
4.3.4 rolling_sum 93
4.3.5 rolling_skew 94
4.3.6 rolling_min 95
4.3.7 rolling_max 96
4.3.8 rolling_median 97
4.3.9 rolling_quantile 98
4.3.10 rolling_rank 99
4.4 exponentially weighted moving functions 99
4.4.1 ewma 99
4.4.2 ewmrms 101
4.4.3 ewmstd 103
4.4.4 ewmvar 104
4.4.5 ewmcor 106
4.4.6 ewmskew 107

4.5 functions exposing their state . . . 108
4.5.1 simple functions . . . 108
4.5.2 expanding window functions . . . 109
4.5.3 rolling window functions . . . 110
4.5.4 exponentially weighted moving functions . . . 110
4.6 Index handling . . . 111
4.6.1 df_fillna . . . 111
4.6.2 df_index . . . 112
4.6.3 df_reindex . . . 113
4.6.4 presync . . . 113
4.6.5 add/sub/mul/div/pow operators . . . 115

5 Tutorials **117**

6 pyg.base.Dict **119**
6.1 initialization . . . 119
6.2 members access . . . 119
6.3 adding . . . 120
6.4 subtracting . . . 120
6.5 modifying the keys: rename . . . 121
6.6 modifying the values: do . . . 121
6.7 Dict can store a calculation flow . . . 121

7 pyg.base.dictable **123**
7.1 Motivation: dictable as an organiser of research flow . . . 123
7.2 Same code, in dictable . . . 124
7.2.1 Oh, no, we have a bad symbol, how do we remove it? . . . 126
7.2.2 Now if we want to calculate something per symbol and window… . . . 126
7.3 dictable functionality . . . 127
7.3.1 construction . . . 127
7.3.2 row access . . . 128
7.3.3 column access . . . 129
7.3.4 d is a dict so supports the usual keys(), values() and items(): . . . 129
7.3.5 column and row access are commutative . . . 130
7.3.6 adding records . . . 130
7.3.7 adding/modifying columns . . . 131
7.3.8 do . . . 132
7.3.9 removing columns . . . 132
7.3.10 removing rows . . . 132
7.3.11 sort . . . 133
7.3.12 listby(keys) . . . 133
7.3.13 unlist . . . 134
7.3.14 groupby(keys) and ungroup . . . 134
7.3.15 inner join . . . 134
7.3.16 inner join (with other columns that match names) . . . 136
7.3.17 cross join . . . 136
7.3.18 xor (versus left and right join) . . . 137
7.3.19 pivot . . . 138
7.3.20 a few observations: . . . 138

8 pyg.mongo **141**
8.1 q . . . 141
8.2 mongo_cursor . . . 143
8.2.1 general objects insertion into documents . . . 143
8.2.2 document reading . . . 144

8.2.3 document writing to files . . . 145
8.2.4 document access . . . 146
8.2.5 filters . . . 147
8.2.6 iteration . . . 147
8.2.7 sorting . . . 148
8.2.8 getitem of a specfic document . . . 148
8.2.9 column access . . . 148
8.2.10 add/remove columns . . . 148
8.2.11 add/remove records . . . 149
8.3 mongo_pk_table . . . 150
8.4 mongo_reader and mongo_pk_reader . . . 152

9 pyg.base.cell 155
9.1 Cell 101 . . . 155
9.2 Workflow without saving to the database . . . 156
9.2.1 some more functions to calculate the profits & loss as well as the signal/noise ratio . . . 158
9.3 Workflow while saving to MongoDB . . . 159
9.3.1 Table creation . . . 159
9.3.2 Any code differences? . . . 159
9.3.3 Accessing the data in MongoDB . . . 161
9.3.4 Calculating the forecasts & saving them . . . 165
9.3.5 Accessing & running the graph once the graph has been created . . . 171
9.4 Comparison of the two workflows . . . 173
9.5 Behind the scene: cell_func . . . 174

10 pyg.base.join 177
10.1 Join . . . 177
10.1.1 Example: Using join function to transfer money to a bank . . . 177
10.1.2 Simple join: inner join between tables . . . 178
10.1.3 Defaults for fields we want to left-join on… . . . 178
10.1.4 Renaming & calculating fields . . . 179
10.2 Perdictable . . . 179
10.2.1 perdictable and caching . . . 181
10.2.2 perdictable with the cell framework . . . 182
10.2.3 perdictable API . . . 184
10.3 Conclusions . . . 185

11 pyg.timeseries 187
11.1 Agreement between pyg.timeseries and pandas . . . 188
11.1.1 Quick performance comparison . . . 188
11.2 pyg and numpy arrays . . . 190
11.3 pandas treatment of nan . . . 191
11.4 pyg.timeseries treatment of nans . . . 192
11.5 Using pyg.timeseries to manage state . . . 193
11.5.1 Example: creating a function exposing its state . . . 194

12 pyg.timeseries decorators 199
12.1 loop . . . 200
12.2 presync: manage indexing and date stamps . . . 201
12.2.1 presync and numpy arrays . . . 202

13 pyg.timeseries.ewma 205
13.1 What happens if the clock does not move at all? . . . 206
13.2 What are valid time parameters? . . . 208

14 Indices and tables 211

Python Module Index 213

Index 215

CHAPTER

ONE

README

- If you examine data by multiple dimensions, you need pyg.base.dictable.
- If you use MongoDB, you need pyg.mongo.
- If you use pandas for timeseries analysis, you should consider using pyg.timeseries.

pyg is both succinct and powerful and makes your code almost boilerplate free and easy to maintain. As an example, I estimate that Man AHL, a leading quant hedge fund, relies on about 50 coders to replicate the functionality and maintain boilerplate code that pyg would make redundant.

Below is autodoc created by sphinx followed by tutorials created in jupyter notebooks.

CHAPTER

TWO

PYG.BASE

2.1 extensions to dict

2.1.1 dictattr

class pyg.base._dictattr.**dictattr**

A simple dict with extended member manipulation

1) access using d.key
2) access multiple elements using d[key1, key2]

Example members access

```
>>> from pyg import *
>>> d = dictattr(a = 1, b = 2, c = 3)
>>> assert isinstance(d, dict)
>>> assert d.a == 1
>>> assert d['a','b'] == [1,2]
>>> assert d[['a','b']] == dictattr(a = 1, b = 2)
```

In addition, it has extended key selection/subsetting

Example subsetting

```
>>> d = dictattr(a = 1, b = 2, c = 3)
>>> assert d - 'a' == dictattr(b = 2, c = 3)
>>> assert d & ['b', 'c', 'not in keys'] == dictattr(b = 2, c = 3)
```

dictattr supports not in-place 'update':

Example updating via adding another dict

```
>>> d = dictattr(a = 1, b = 2) + dict(b = 'replacing old value', c = 'new key')
>>> assert d == dictattr(a = 1, b = 'replacing old value', c = 'new key')
```

copy() → a shallow copy of D

keys()

dictattr returns an actual list rather than a generator. Further, this recognises that the keys are necessarily unique so it returns a ulist which is also a set

Returns

ulist list of keys of dictattr.

Example

```
>>> from pyg import *
>>> d = dictattr(a = 1, b = 2)
>>> assert d.keys() == ulist(['a', 'b'])
>>> assert d.keys() & ['a', 'c', 'd'] == ['a']
```

relabel (**args, **relabels*)
Identical to relabel. See relabel for full docs

rename (**args, **relabels*)
Identical to relabel. See relabel for full docs

values () → an object providing a view on D's values

pyg.base._dictattr.dictattr.**__add__**(*self, other*)
dictattr uses add as a copy + update. Similar to the latest python |=

Example

```
>>> from pyg import *
>>> d = dictattr(a = 1, b = 2)
>>> assert d + dict(b = 3, c = 5) == dictattr(a = 1, b = 3, c = 5)
```

Parameters

other: dict a dict used to update current dict.

pyg.base._dictattr.dictattr.**__sub__**(*self, key, copy=True*)
deletes an item but does not throw an exception if not there dictattr uses subtraction to remove key(s)

Returns

updated dictattr

Example

```
>>> from pyg import *
>>> d = dictattr(a = 1, b = 2, c = 3)
>>> assert d - ['b','c'] == dictattr(a = 1)
>>> assert d - 'c' == dictattr(a = 1, b = 2)
>>> assert d - 'key not there' == d
>>> #commutative
>>> assert (d - 'c').keys() == d.keys() - 'c'
```

pyg.base._dictattr.dictattr.**__and__**(*self, other*)
dictattr uses & as a set operator for key filtering

Returns

updated dictattr

Example

```
>>> from pyg import *
>>> d = dictattr(a = 1, b = 2, c = 3)
>>> assert d & ['a', 'b', 'not_there'] == dictattr(a = 1, b = 2)
```

(continues on next page)

(continued from previous page)

```
>>> #commutative
>>> assert (d & ['a', 'b', 'x']).keys() == d.keys() & ['a', 'b', 'x']
```

2.1.2 ulist

The dictattr.keys() method returns a ulist: a list with unique elements:

class pyg.base._ulist.**ulist**(**args*, *unique=False*)

A list whose members are unique. It has +/- operations overloaded while also supporting set opeations &/|

Example

```
>>> assert ulist([1,3,2,1]) == list([1,3,2])
```

Example addition adds element(s)

```
>>> assert ulist([1,3,2,1]) + 4  == list([1,3,2,4])
>>> assert ulist([1,3,2,1]) + [4,1] == list([1,3,2,4])
>>> assert ulist([1,3,2,1]) + [4,1,5] == list([1,3,2,4,5])
```

Example subtraction removes element(s)

```
>>> assert ulist([1,3,2,1]) - 1 == [3,2]
>>> assert ulist([1,3,2,1]) - [1,3,4] == [2]
```

Example set operations

```
>>> assert ulist([1,3,2,1]) & 1 == [1]
>>> assert ulist([1,3,2,1]) & [1,3,4] == [1,3]
```

```
>>> assert ulist([1,3,2,1]) | 1 == [1,3,2]
>>> assert ulist([1,3,2,1]) | 4 == [1,3,2,4]
>>> assert ulist([1,3,2,1]) | [1,3,4] == [1,3,2,4]
```

copy()

Return a shallow copy of the list.

2.1.3 Dict

class pyg.base._dict.**Dict**

Dict extends dictattr to allow access to *functions* of members

Example

```
>>> from pyg import *
>>> d = Dict(a = 1, b=2)
>>> assert d[lambda a, b: a+b] == 3
>>> assert d['a','b', lambda a,b: a+b] == [1,2,3]
```

Dict is also callable where the key-value is used to add/update current members

Example

```
>>> from pyg import *
>>> d = Dict(a = 1, b=2)
>>> assert d(c = 3) == Dict(a = 1, b = 2, c = 3)
>>> assert d(c = lambda a,b: a+b) == Dict(a = 1, b = 2, c = 3)
```

```
>>> assert d(c = 3) == Dict(a = 1, b = 2) + Dict(c = 3)
>>> assert Dict(a = 1)(b = lambda a: a+1)(c = lambda a,b: a+b) == Dict(a = 1,b =
↪2,c = 3)
```

do (*function*, **keys*)
applies a function(s) on multiple keys at the same time

Parameters

function [callable or list of callables] function to be applied per column

***keys** [string/list of strings] list of columns to be applied. If missing, applied to all columns

Returns

res : Dict

Example

```
>>> from pyg import *
>>> d = Dict(name = 'adam', surname = 'atkins')
>>> assert d.do(proper) == Dict(name = 'Adam', surname = 'Atkins')
```

Example using another key in the calculation

```
>>> from pyg import *
>>> d = Dict(a = 1, b = 5, denominator = 10)
>>> d = d.do(lambda value, denominator: value/denominator, 'a', 'b')
>>> assert d == Dict(a = 0.1, b = 0.5, denominator = 10)
```

pyg.base._dict.Dict.**__call__**(*self*, ***kwargs*)
Call self as a function.

2.1.4 dictable

class pyg.base._dictable.**dictable**(*data=None*, *columns=None*, ***kwargs*)

What is dictable?

dictable is a table, a collection of iterable records. It is also a dict with each key being a column. Why not use a pandas.DataFrame? pd.DataFrame leads a dual life:

- by day an index-based optimized numpy array supporting e.g. timeseries analytics etc.
- by night, a table with keys supporting filtering, aggregating, pivoting on keys as well as inner/outer joining on keys.

dictable only tries to do the latter. dictable should be thought of as a 'container for complicated objects' rather than just an array of primitive floats. In general, each cell may contain timeseries, yield_curves, machine-learning experiments etc. The interface is very succinct and allows the user to concentrate on logic of the calculations rather than boilerplate.

dictable supports quite a flexible construction:

Example construction using records

```
>>> from pyg import *; import pandas as pd
>>> d = dictable([dict(name = 'alan', surname = 'atkins', age = 39, country = 'UK
↪'),
>>>               dict(name = 'barbara', surname = 'brown', age = 29, country =
↪'UK')])
```

Example construction using columns and constants

```
>>> d = dictable(name = ['alan', 'barbara'], surname = ['atkins', 'brown'], age =␣
↪[39, 29], country = 'UK')
```

Example construction using pandas.DataFrame

```
>>> original = dictable(name = ['alan', 'barbara'], surname = ['atkins', 'brown'],
↪ age = [39, 29], country = 'UK')
>>> df_from_dictable = pd.DataFrame(original)
>>> dictable_from_df = dictable(df_from_dictable)
>>> assert original == dictable_from_df
```

Example construction rows and columns

```
>>> d = dictable([['alan', 'atkins', 39, 'UK'], ['barbara', 'brown', 29, 'UK']],␣
↪columns = ['name', 'surname', 'age', 'country'])
```

Access column access

```
>>> assert d.keys() ==  ['name', 'surname', 'age', 'country']
>>> assert d.name == ['alan', 'barbara']
>>> assert d['name'] == ['alan', 'barbara']
>>> assert d['name', 'surname'] == [('alan', 'atkins'), ('barbara', 'brown')]
>>> assert d[lambda name, surname: '%s %s'%(name, surname)] == ['alan atkins',
↪'barbara brown']
```

Access row access & iteration

```
>>> assert d[0] == {'name': 'alan', 'surname': 'atkins', 'age': 39, 'country': 'UK
↪'}
>>> assert [row for row in d] == [{'name': 'alan', 'surname': 'atkins', 'age': 39,
↪ 'country': 'UK'},
>>>                               {'name': 'barbara', 'surname': 'brown', 'age':␣
↪29, 'country': 'UK'}]
```

Note that members access is commutative:

```
>>> assert d.name[0] == d[0].name == 'alan'
>>> d[lambda name, surname: name + surname][0] == d[0][lambda name, surname: name␣
↪+ surname]
>>> assert sum([row for row in d], dictable()) == d
```

Example adding records

```
>>> d = dictable(name = ['alan', 'barbara'], surname = ['atkins', 'brown'], age =␣
↪[39, 29], country = 'UK')
>>> d = d + {'name': 'charlie', 'surname': 'chocolate', 'age': 49} # can add a␣
↪record directly
>>> assert d[-1] == {'name': 'charlie', 'surname': 'chocolate', 'age': 49,
↪'country': None}
>>> d += dictable(name = ['dana', 'ender'], surname = ['deutch', 'esterhase'],␣
↪age = [10, 20], country = ['Germany', 'Hungary'])
>>> assert d.name == ['alan', 'barbara', 'charlie', 'dana', 'ender']
>>> assert len(dictable.concat([d,d])) == len(d) * 2
```

Example adding columns

```
>>> d = dictable(name = ['alan', 'barbara'], surname = ['atkins', 'brown'], age =␣
↪[39, 29], country = 'UK')
```

```
>>> ### all of the below are ways of adding columns ####
>>> d.gender == ['m', 'f']
>>> d = d(gender = ['m', 'f'])
>>> d['gender'] == ['m', 'f']
>>> d2 = dictable(gender = ['m', 'f'], profession = ['astronaut', 'barber'])
>>> d = d(**d2)
```

Example adding derived columns

```
>>> d = dictable(name = ['alan', 'barbara'], surname = ['atkins', 'brown'], age =␣
↪[39, 29], country = 'UK')
>>> d = d(full_name = lambda name, surname: proper('%s %s'%(name, surname)))
>>> d['full_name'] = d[lambda name, surname: proper('%s %s'%(name, surname))]
>>> assert d.full_name == ['Alan Atkins', 'Barbara Brown']
```

Example dropping columns

```
>>> d = dictable(name = ['alan', 'barbara'], surname = ['atkins', 'brown'], age =␣
↪[39, 29], country = 'UK')
>>> del d.country # in place
>>> del d['age'] # in place
>>> assert (d - 'name')[0] ==  {'surname': 'atkins'} and d[0] == {'name': 'alan',
↪'surname': 'atkins'}
```

Example row selection, inc/exc

```
>>> d = dictable(name = ['alan', 'barbara'], surname = ['atkins', 'brown'], age =␣
↪[39, 29], country = 'UK')
>>> assert len(d.exc(name = 'alan')) == 1
>>> assert len(d.exc(lambda age: age<30)) == 1 # can filter on *functions* of␣
↪members, not just members.
>>> assert d.inc(name = 'alan').surname == ['atkins']
>>> assert d.inc(lambda age: age<30).name == ['barbara']
>>> assert d.exc(lambda age: age<30).name == ['alan']
```

dictable supports:

- sort
- group-by/ungroup
- list-by/ unlist
- pivot/unpivot
- inner join, outer join and xor

Full details are below.

classmethod concat (**others*)

adds together multiple dictables. equivalent to sum(others, self) but a little faster

Parameters

***others** [dictables] records to be added to current table

Returns

merged [dictable] sum of all records

Example

```
>>> from pyg import *
>>> d1 = dictable(a = [1,2,3])
>>> d2 = dictable(a = [4,5,6])
>>> d3 = dictable(a = [7,8,9])
```

```
>>> assert dictable.concat(d1,d2,d3) == dictable(a = range(1,10))
>>> assert dictable.concat([d1,d2,d3]) == dictable(a = range(1,10))
```

do (*function*, **keys*)

applies a function(s) on multiple keys at the same time

Parameters

function [callable or list of callables] function to be applied per column

***keys** [string/list of strings] list of columns to be applied. If missing, applied to all columns

Returns

res : dictable

Example

```
>>> from pyg import *
>>> d = dictable(name = ['adam', 'barbara', 'chris'], surname = ['atkins',
↪'brown', 'cohen'])
>>> assert d.do(proper) == dictable(name = ['Adam', 'Barbara', 'Chris'],␣
↪surname = ['Atkins', 'Brown', 'Cohen'])
```

Example using another column in the calculation

```
>>> from pyg import *
>>> d = dictable(a = [1,2,3,4], b = [5,6,9,8], denominator = [10,20,30,40])
>>> d = d.do(lambda value, denominator: value/denominator, 'a', 'b')
>>> assert d == dictable(a = 0.1, b = [0.5,0.3,0.3,0.2], denominator = [10,20,
↪30,40])
```

exc (**functions*, ***filters*)

performs a filter on what rows to exclude

Parameters

***functions** [callables or a dict] filters based on functions of each row

****filters** [value or list of values] filters per each column

Returns

dictable table with rows that satisfy all conditions excluded.

Example filtering on keys

```
>>> from pyg import *; import numpy as np
>>> d = dictable(x = [1,2,3,np.nan], y = [0,4,3,5])
>>> assert d.exc(x = np.nan) == dictable(x = [1,2,3], y = [0,4,3])
>>> assert d.exc(x = 1) == dictable(x = [2,3,np.nan], y = [4,3,5])
>>> assert d.exc(x = [1,2]) == dictable(x = [1,2], y = [0,4])
```

Example filtering on callables

```
>>> from pyg import *; import numpy as np
>>> d = dictable(x = [1,2,3,np.nan], y = [0,4,3,5])
>>> assert d.exc(lambda x,y: x>y) == dictable(x = 1, y = 0)
```

get (*key*, *default=None*)

Return the value for key if key is in the dictionary, else default.

groupby (**by*, *grp='grp'*)

Similar to pandas groupby but returns a dictable of dictables with a new column ‘grp’

Example

```
>>> x = dictable(a = [1,2,3,4], b= [1,0,1,0])
>>> res = x.groupby('b')
>>> assert res.keys() == ['b', 'grp']
>>> assert is_dictable(res[0].grp) and res[0].grp.keys() == ['a']
```

Parameters

*by : str or list of strings

gr.

grp [str, optional] The name of the column for the dictables per each key. The default is ‘grp’.

Returns

dictable A dictable containing the original keys and a dictable per unique key.

`inc` (**functions*, ***filters*)
performs a filter on what rows to include

Parameters

***functions** [callables or a dict] filters based on functions of each row

****filters** [value or list of values] filters per each column

Returns

dictable table with rows that satisfy all conditions.

Example filtering on keys

```
>>> from pyg import *; import numpy as np
>>> d = dictable(x = [1,2,3,np.nan], y = [0,4,3,5])
>>> assert d.inc(x = np.nan) == dictable(x = np.nan, y = 5)
>>> assert d.inc(x = 1) == dictable(x = 1, y = 0)
>>> assert d.inc(x = [1,2]) == dictable(x = [1,2], y = [0,4])
```

Example filtering on callables

```
>>> from pyg import *; import numpy as np
>>> d = dictable(x = [1,2,3,np.nan], y = [0,4,3,5])
>>> assert d.inc(lambda x,y: x>y) == dictable(x = 1, y = 0)
```

`join` (*other*, *lcols=None*, *rcols=None*, *mode=None*)
Performs either an inner join or a cross join between two dictables

Example inner join

```
>>> from pyg import *
>>> x = dictable(a = ['a','b','c','a'])
>>> y = dictable(a = ['a','y','z'])
>>> assert x.join(y) == dictable(a = ['a', 'a'])
```

Example outer join

```
>>> from pyg import *
>>> x = dictable(a = ['a','b'])
>>> y = dictable(b = ['x','y'])
>>> assert x.join(y) == dictable(a = ['a', 'a', 'b', 'b'], b = ['x', 'y', 'x',
↪ 'y'])
```

`pivot` (*x*, *y*, *z*, *agg=None*)
pivot table functionality.

Parameters

x [str/list of str] unique keys per each row

y [str] unique key per each column

z [str/callable] A column in the table or an evaluated quantity per each row

agg [None/callable or list of callables, optional] Each (x,y) cell can potentially contain multiple z values. so if agg = None, a list is returned If you want the data aggregated in any way, then supply an aggregating function(s)

Returns

A dictable which is a pivot table of the original data

Example

```
>>> from pyg import *
>>> timetable_as_list = dictable(x = [1,2,3]) * dictable(y = [1,2,3])
>>> timetable = timetable_as_list.xyz('x','y',lambda x, y: x * y)
>>> assert timetable = dictable(x = [1,2,3], )
```

Example

```
>>> self = dictable(x = [1,2,3]) * dictable(y = [1,2,3])
>>> x = 'x'; y = 'y'; z = lambda x, y: x * y
>>> self.exc(lambda x, y: x+y==5).xyz(x,y,z, len)
```

sort (**by*)
Sorts the table either using a key, list of keys or functions of members

Example

```
>>> import numpy as np
>>> self = dictable(a = [_ for _ in 'abracadabra'], b=range(11), c = range(0,
↪33,3))
>>> self.d = list(np.array(self.c) % 11)
>>> res = self.sort('a', 'd')
>>> assert list(res.c) == list(range(11))
```

```
>>> d = dictable(a = ['a', 1, 'c', 0, 'b', 2]).sort('a')
>>> res = d.sort('a','c')
>>> print(res)
>>> assert ''.join(res.a) == 'aaaaabbcdrr' and list(res.c) == [0,4,8,9,10] +␣
↪[2,3] + [1] + [7] + [5,6]
```

```
>>> d = d.sort(lambda b: b*3 % 11) ## sorting again by c but using a function
>>> assert list(d.c) == list(range(11))
```

ungroup (*grp='grp'*)
Undoes groupby

Example

```
>>> x = dictable(a = [1,2,3,4], b= [1,0,1,0])
>>> self = x.groupby('b')
```

Parameters

grp [str, optional] column name where dictables are. The default is 'grp'.

Returns

dictable.

unlist()
undoes listby…

Example

```
>>> x = dictable(a = [1,2,3,4], b= [1,0,1,0])
>>> x.listby('b')
```

dictable[2 x 2] b|a 0|[2, 4] 1|[1, 3]

```
>>> assert x.listby('b').unlist().sort('a') == x
```

Returns

dictable a dictable where all rows with list in them have been 'expanded'.

unpivot(*x*, *y*, *z*)
undoes self.xyz / self.pivot

Example

```
>>> from pyg import *
>>> orig = (dictable(x = [1,2,3,4]) * dict(y = [1,2,3,4,5]))(z = lambda x, y:␣
↪x*y)
>>> pivot = orig.xyz('x', 'y', 'z', last)
>>> unpivot = pivot.unpivot('x','y','z').do(int, 'y') # the conversion to␣
↪column names mean y is now string... so we convert back to int
>>> assert orig == unpivot
```

Parameters

x [str/list of strings] list of keys in the pivot table.

y [str] name of the columns that wil be used for the values that are currently column headers.

z [str] name of the column that describes the data currently within the pivot table.

Returns

dictable

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

xor(*other*, *lcols=None*, *rcols=None*, *mode='l'*)
returns what is in lhs but NOT in rhs (or vice versa if mode = 'r'). Together with inner joining, can be used as left/right join

Examples

```
>>> from pyg import *
>>> self = dictable(a = [1,2,3,4])
>>> other = dictable(a = [1,2,3,5])
>>> assert self.xor(other) == dictable(a = 4) # this is in lhs but not in rhs
>>> assert self.xor(other, lcols = lambda a: a * 2, rcols = 'a') ==␣
↪dictable(a = [2,3,4]) # fit can be done using formulae rather than actual␣
↪columns
```

(continues on next page)

(continued from previous page)

The XOR functionality can be performed using quotient (divide): >>> assert lhs/rhs == dictable(a = 4) >>> assert rhs/lhs == dictable(a = 5)

```
>>> rhs = dictable(a = [1,2], b = [3,4])
>>> left_join_can_be_done_simply_as = lhs * rhs + lhs/rhs
```

Parameters

other [dictable (or something that can be turned to one)] what we exclude with.

lcols [str/list of strs, optional] the left columns/formulae on which we match. The default is None.

rcols [str/list of strs, optional] the right columns/formulae on which we match. The default is None.

mode [string, optional] When set to 'r', performs xor the other way. The default is 'l'.

Returns

dictable a dictable containing what is in self but not in ther other dictable.

xyz (*x*, *y*, *z*, *agg=None*)
pivot table functionality.

Parameters

x [str/list of str] unique keys per each row

y [str] unique key per each column

z [str/callable] A column in the table or an evaluated quantity per each row

agg [None/callable or list of callables, optional] Each (x,y) cell can potentially contain multiple z values. so if agg = None, a list is returned If you want the data aggregated in any way, then supply an aggregating function(s)

Returns

A dictable which is a pivot table of the original data

Example

```
>>> from pyg import *
>>> timetable_as_list = dictable(x = [1,2,3]) * dictable(y = [1,2,3])
>>> timetable = timetable_as_list.xyz('x','y',lambda x, y: x * y)
>>> assert timetable = dictable(x = [1,2,3], )
```

Example

```
>>> self = dictable(x = [1,2,3]) * dictable(y = [1,2,3])
>>> x = 'x'; y = 'y'; z = lambda x, y: x * y
>>> self.exc(lambda x, y: x+y==5).xyz(x,y,z, len)
```

pyg.base._dictable.dictable.**__call__**(*self*, ***kwargs*)
Call self as a function.

2.1.5 perdictable

pyg.base._perdictable.**perdictable**()

A decorator that makes a function works per dictable and not just on original value

Example

```
>>> f = lambda a, b: a+b
>>> p = perdictable(f, on = ['key'])
```

The new modified function p now works the same on old values:

Paramaters

function [callable] A function

on: str/list of str perform join based on these keys

renames: dict This tells us which column to grab from which table

defaults: dict If a default is provided for a parameter, we will perform a left join, substituting missing values with the defaults

if_none: bool, list of keys If historic data is None while the row has expired, should we force a recalculation? if True, will be done.

output_is_input: bool, list of keys Some functions want their own outut to be presented to them. If you see to True, if cached values exist for these columns, these are provided to the function

include_inputs: When we return the outputs, do you want the inputs to be included as well in the dictable.

col: str the name of the variable output.

Example

```
>>> f = lambda a, b: a+b
>>> p = perdictable(f, include_inputs = True)
>>> assert p(a = 1, b = 2) == 3
>>> assert p(a = dictable(a = [1,2,3]), b = 3) == dictable(a = [1,2,3], b = 3,␣
↪expiry = None, data = [4,5,6])
```

some parameters are constant, some are tables…

```
>>> assert p(a = 1, b = dictable(key = ['a','b','c'], b = [1,2,3])) ==␣
↪dictable(key  = ['a', 'b', 'c'], data = [2,3,4])
```

multiple tables… some unkeyed

```
>>> assert p(a = dictable(a = [1,2]), b = dictable(key = ['a','b','c'], b = [1,2,
↪3])) == dictable(key  = ['a','a', 'b', 'b', 'c','c'], data = [2,3,3,4,4,5])
```

multiple tables… all keyed

```
>>> a = dictable(key = ['x', 'y'], data = [1,2])
>>> b = dictable(key = ['y', 'z'], data = [3,4])
>>> assert p(a = a, b = b) == dictable(key  = ['y'], data = [5])
```

Example existing data provided using data and expiry

```
>>> a = dictable(key = ['x', 'y', 'z'], data = [1,2,3])
>>> b = dictable(key = ['x', 'y', 'z'], data = [1,3,4])
>>> data = dictable(key = ['x', 'y'], data = ['we calculated this before', 'we␣
↪calculated before but hasnt expired'])
>>> expiry = dictable(key = ['x', 'y'], data = [dt(2000,1,1), dt(3000,1,1)])
>>> inputs = dict(a = a, b = b)
```

```
>>> res = p(a = a, b = b, data = data, expiry = expiry)
>>> assert res.find_data(key = 'x').data == 'we calculated this before'
>>> assert res.find_data(key = 'y').data == 5  # although calculated before, we␣
↪recalculate as its expiry is in the future
```

function = lambda a, b: dict(sum = a+b, prod = a*b); function.output = ['sum', 'prod'] self = perdictable(function) self(a = 1, b = 2) inputs = dict(a = dictable(a = [1,2,3]), b = 2); expiry = None self(a = dictable(a = [1,2,3]), b = 2)

2.1.6 join

`pyg.base._perdictable.join`(*inputs*, *on=None*, *renames=None*, *defaults=None*)

Suppose we have a function which is defined on simple numbers

Example

```
>>> from pyg import *
>>> profit = lambda amount, price: amount  * price
```

The amounts sold are available in one table and prices in another

Example

```
>>> amounts = dictable(product = ['apple', 'orange', 'pear'], amount = [1,2,3])
>>> prices = dictable(product = ['apple', 'orange', 'pear', 'banana'], price = [4,
↪5,6,8])
>>> join(dict(amount = amounts, price = prices), on = 'product')(profit = profit)
```

```
>>> dictable[3 x 4]
>>> product|amount|price|profit
>>> apple  |1     |4    |4
>>> orange |2     |5    |10
>>> pear   |3     |6    |18
```

Parameters

inputs [dict] a dict of input parameters, some of them may be dictables.

on [str/list of str] when we have dictables

renames [dict, optional] remapping. if the datasets contain multiple columns, you can say renames = dict(price = 'price_in_dollar') to tell the algo, this is the column to use The default is None.

defaults [dict, optional] Normally, an inner join is performed. However, if there is a default value/formula for when e.g. a price is missing, use this. The default is None.

Returns

dictable a dictable of an inner join.

Example how column mapping is done

```
>>> on = 'a'
>>> ## if there is only one column apart from keys, then it is selected:
```

```
>>> assert join(dict(x = dictable(a = [1,2], data = [2,3])), on = on) ==␣
↪dictable(a = [1,2], x = [2,3])
>>> assert join(dict(x = dictable(a = [1,2], random_name = [2,3])), on = on) ==␣
↪dictable(a = [1,2], x = [2,3])
```

```
>>> ## if there are multiple columns, if variable name is there, we use it:
>>> assert join(dict(x = dictable(a = [1,2], z = [2,3], x = [4,5])), on) ==␣
↪dictable(a = [1,2], x = [4,5])
```

```
>>> ## if there are multiple columns, and 'data' is one of the columns, we use it:
>>> assert join(dict(x = dictable(a = [1,2], z = [2,3], data = [4,5])), on) ==␣
↪dictable(a = [1,2], x = [4,5])
```

Example how column mapping is done with rename

```
>>> with pytest.raises(KeyError):
>>>     join(dict(x = dictable(a = [1,2], b = [2,3], c = [4,5])), on = 'a') ##␣
↪pick b or c?
>>> assert join(dict(x = dictable(a = [1,2], b = [2,3], c = [4,5])), on = 'a',␣
↪renames = dict(x = 'c')) == dictable(a = [1,2,], x = [4, 5])
```

Example joins with partial columns in some tables

```
>>> on = ['a', 'b', 'c']
>>> a = dictable(a = [1,2,3,4], x = [1,2,3,4]) ## only column a here
>>> b = dictable(b = [1,2,3,4], y = [1,2,3,4]) ## only column b here
>>> c = dictable(a = [1,2,3,4], b = [1,2,3,4], c = [1,2,3,4], z = [1,2,3,4])
>>> j = join(dict(x = a, y = b, z = c), on = ['a', 'b', 'c'])
>>> assert len(j) == 4 and sorted(j.keys()) == ['a', 'b', 'c', 'x', 'y', 'z']
```

Example join with defaults

If no defaults are provided, we need all variables to be present. However, if we specify defaults, we left-join on that variable and insert the default value

```
>>> x = dictable(a = [1,2,4], x = [1,2,4])
>>> y = dictable(a = [1,2,3], x = [5,6,7])
>>> on = 'a'
>>> assert join(dict(x = x, y = y), on = on) == dictable(a = [1,2,], x = [1,2], y␣
↪= [5,6])
>>> assert join(dict(x = x, y = y), on = 'a', defaults = dict(x = None)) ==␣
↪dictable(a = [1,2,3], x = [1,2,None], y = [5,6,7])
>>> assert join(dict(x = x, y = y), on = 'a', defaults = dict(y = 0)) ==␣
↪dictable(a = [1,2,4], x = [1,2,4], y = [5,6,0])
>>> assert join(dict(x = x, y = y), on = 'a', defaults = dict(x = None, y = 0))␣
↪== dictable(a = [1,2,3,4], x = [1,2,None,4], y = [5,6,7,0])
```

2.1.7 named_dict

`pyg.base._named_dict.named_dict`(*name*, *keys*, *defaults={}*, *types={}*, *casts={}*, *basedict='pyg.base.dictattr'*, *debug=False*)

This forms a base for all classes. It is similar to named_tuple but:

- supports additional features such as casting/type checking.
- support default values

The resulting class is a dict so can be stored in MongoDB, sent to json or be used to construct a pd.Series automatically.

Example Simple construction

```
>>> Customer = named_dict('Customer', ['name', 'date', 'balance'])
>>> james = Customer('james', 'today', 10)
>>> assert james['balance'] == 10
>>> assert james.date == 'today'
```

Example How named_dict works with json/pandas/other named_dicts

```
>>> class Customer(named_dict('Customer', ['name', 'date', 'balance'])):
>>>     def add_to_balance(self, value):
>>>         res = self.copy()
>>>         res.balance += value
>>>         return res
```

```
>>> james = Customer('james', 'date', 10)
>>> assert james.add_to_balance(10).balance == 20
>>> import json
>>> assert pd.Series(james).date == 'date'
>>> assert dict(james) == {'name': 'james', 'date': 'date', 'balance': 10}
>>> assert json.dumps(james) == '{"name": "james", "date": "date", "balance": 10}'
```

```
>>> class VIP(named_dict('VIP', ['name', 'date'])):
>>>     def some_method(self):
>>>         return 'inheritence between classes works as long as members can share
↪'
```

```
>>> vip = VIP(james)
>>> assert vip.name == 'james' ## members moved seemlessly
>>> assert vip.some_method() == 'inheritence between classes works as long as␣
↪members can share'
```

Example Adding defaults

```
>>> Customer = named_dict('Customer', ['name', 'date', 'balance'], defaults =␣
↪dict(balance = 0))
>>> james = Customer('james', 'today')
>>> assert james['balance'] == 0
```

Example types checking

```
>>> import datetime
>>> Customer = named_dict('Customer', ['name', 'date', 'balance'], defaults =␣
↪dict(balance = 0), types = dict(date = 'datetime.datetime'))
>>> james = Customer('james', datetime.datetime.now())
>>> assert james['balance'] == 0
```

Example casting

```
>>> Customer = named_dict('Customer', ['name', 'date', 'balance'], defaults =␣
↪dict(balance = 0), types = dict(date = 'datetime.datetime'), casts =␣
↪dict(balance = 'float'))
>>> james = Customer('james', datetime.datetime.now(), balance = '10.3')
>>> assert james['balance'] == 10.3
```

Parameters

name [str] name of new class.

keys [list] list of keys that the class must have as members.

defaults [dict, optional] default values for the keys. The default is {}.

types [type or callable, optional] A test to be applied for keys either as a callable or as a type. The default is {}.

casts [dict, optional] function. The default is {}.

basedict [str, optional] name of the dict class to inherit from. The default is 'dict'.

debug [bool, optional] output the construction text if set to True. The default is False.

ValueError DESCRIPTION.

Returns

result : new class that inherits from a dict

2.2 decorators

2.2.1 wrapper

class pyg.base._decorators.**wrapper**(*function=None*, **args*, ***kwargs*)

A base class for all decorators. It is similar to functools.wraps but better. You basically need to define the wrapped method and everything else is handled for you. - You can then use it either directly to decorate functions - Or use it to create parameterized decorators - the __name__, __wrapped__, __doc__ and the getargspec will all be taken care of.

Example

```
>>> class and_add(wrapper):
>>>     def wrapped(self, *args, **kwargs):
>>>         return self.function(*args, **kwargs) + self.add ## note that we are␣
↪assuming self.add exists
```

```
>>> @and_add(add = 3) ## create a decorator and decorate the function
>>> def f(a,b):
>>>     return a+b
```

```
>>> assert f.add == 3
>>> assert f(1,2) == 6
```

Alternatively you can also use it this directly:

```
>>> def f(a,b):
>>>     return a+b
>>>
>>> assert and_add(f, add = 3)(1,2) == 6
```

Example Explicit parameter construction

You can make the init more explict, also adding defaults for the parameters:

```
>>> class and_add_version_2(wrapper):
>>>     def __init__(self, function = None, add = 3):
>>>         super(and_add, self).__init__(function = function, add = add)
>>>     def wrapped(self, *args, **kwargs):
>>>         return self.function(*args, **kwargs) + self.add
```

```
>>> @and_add_version_2
>>> def f(a,b):
>>>     return a+b
>>> assert f(1,2) == 6
```

The decorator is designed to have a single instance of a specific wrapper

```
>>> f = lambda a, b: a+b
>>> assert and_add(and_add(f)) == and_add(f)
```

This holds even for multiple levels of wrapping:

```
>>> x = try_none(and_add(f))
>>> y = try_none(and_add(x))
>>> assert x == y
>>> assert x(1, 'no can add') is None
```

2.2.2 timer

class pyg.base._decorators.**timer**(*function*, *n=1*, *time=False*)

timer is similar to timeit but rather than execution of a Python statement, timer wraps a function to make it log its evaluation time before returning output

Parameters

function: callable The function to be wraooed

n: int, optional Number of times the function is to be evaluated. Default is 1

time: bool, optional If set to True, function will return the TIME it took to evaluate rather than the original function output.

Example

```
>>> from pyg import *; import datetime
>>> f = lambda a, b: a+b
>>> evaluate_100 = timer(f, n = 100, time = True)(1,2)
>>> evaluate_10000 = timer(f, n = 10000, time = True)(1,2)
>>> assert evaluate_10000> evaluate_100
>>> assert isinstance(evaluation_time, datetime.timedelta)
```

2.2.3 try_value

pyg.base._decorators.**try_value**()

wraps a function to try an evaluation. If an exception is thrown, returns a cached argument

Parameters

function callable The function we want to decorate

value: If the function fails, it will return value instead. Default is None

verbose: bool If set to True, the logger will warn with the error message.

There are various convenience functions with specific values try_zero, try_false, try_true, try_nan and try_none will all return specific values if function fails.

Example

```
>>> from pyg import *
>>> f = lambda a: a[0]
>>> assert try_none(f)(4) is None
>>> assert try_none(f, 'failed')(4) == 'failed'
```

2.2.4 try_back

pyg.base._decorators.**try_back**()

wraps a function to try an evaluation. If an exception is thrown, returns first argument

Example

```
>>> f = lambda a: a[0]
>>> assert try_back(f)('hello') == 'h' and try_back(f)(5) == 5
```

2.2.5 loops

class pyg.base._loop.**loops**(*function=None*, *types=None*)

converts a function to loop over the arguments, depending on the type of the first argument

Examples

```
>>> @loop(dict, list, pd.DataFrame, pd.Series)
>>> def f(a,b):
>>>     return a+b
```

```
>>> assert f(1,2) == 3
>>> assert f([1,2,3],2) == [3,4,5]
>>> assert f([1,2,3], [4,5,6]) == [5,7,9]
```

```
>>> assert f(dict(x=1,y=2), 3) == dict(x = 4, y = 5)
>>> assert f(dict(x=1,y=2), dict(x = 3, y = 4)) == dict(x = 4, y = 6)
```

```
>>> a = pd.Series(dict(x=1,y=2))
>>> b = dict(x=3,y=4)
>>> assert np.all(f(a,b) == pd.Series(dict(x=4,y=6)))
```

```
>>> a = pd.DataFrame(dict(x=[1,1],y=[2,2])); a.index = [5,10]
>>> b = dict(x=3,y=4)
>>> res =  f(a,b)
>>> assert np.all(res == pd.DataFrame(dict(x=[4,4],y=[6,6]), index = [5,10]))
```

```
>>> a = pd.DataFrame(dict(x=[1,1],y=[2,2])); a.index = [5,10]
>>> res =  f(a,[3,4])
>>> assert np.all( res == pd.DataFrame(dict(x=[4,4],y=[6,6]), index = [5,10]))
```

2.2.6 loop

`pyg.base._loop.`**`loop`**`(`**types*`)`

returns an instance of loops(types = types)

loop_all is an instance of loops that loops over dict, list, tuple, np.ndarray and pandas.DataFrame/Series

2.2.7 kwargs_support

`pyg.base._decorators.`**`kwargs_support`**`()`

Extends a function to support **kwargs inputs

Example

```
>>> from pyg import *
>>> @kwargs_support
>>> def f(a,b):
>>>     return a+b
```

```
>>> assert f(1,2, what_is_this = 3, not_used = 4, ignore_this_too = 5) == 3
```

2.3 graphs & cells

2.3.1 cell

`class` `pyg.base._cell.`**`cell`**`(`*function=None*, *output=None*, ***kwargs*`)`

cell is a Dict that can be though of as a node in a calculation graph. The nearest parallel is actually an Excel cell:

- cell contains both its function and its output. cell.output defines the keys where the output is supposed to be

- cell contains reference to all the function outputs
- cell contains its locations and the means to manage its own persistency

Parameters

- function is the function to be called
- ** kwargs are the function named key value args. NOTE: NO SUPPORT for *args nor **kwargs in function
- output: where should the function output go?

Example simple construction

```
>>> from pyg import *
>>> c = cell(lambda a, b: a+b, a = 1, b = 2)
>>> assert c.a == 1
>>> c = c.go()
>>> assert c.output == ['data'] and c.data == 3
```

Example make output go to 'value' key

```
>>> c = cell(lambda a, b: a+b, a = 1, b = 2, output = 'value')
>>> assert c.go().value == 3
```

Example multiple outputs by function

```
>>> f = lambda a, b: dict(sum = a+b, prod = a*b)
>>> c = cell(f, a = 1, b = 2, output  = ['sum', 'prod'])
>>> c = c.go()
>>> assert c.sum == 3 and c.prod == 2
```

Methods

- cell.run() returns bool if cell needs to be run
- cell.go() calculates the cell and returns the function with cell.output keys now populated.
- cell.load()/cell.save() interface for self load/save persistence

`copy()` → a shallow copy of D

2.3.2 cell_go

`pyg.base._cell.cell_go`(*value*, *go=1*)

cell_go makes a cell run (using cell.go(go)) and returns the calculated cell. If value is not a cell, value is returned.

Parameters

value [cell] The cell (or anything else).

go [int] same inputs as per cell.go(go). 0: run if cell.run() is True 1: run this cell regardless, run parent cells only if they need to calculate too n: run this cell & its nth parents regardless.

Returns

The calculated cell

Example calling non-cells

```
>>> assert cell_go(1) == 1
>>> assert cell_go(dict(a=1,b=2)) == dict(a=1,b=2)
```

Example calling cells

```
>>> c = cell(lambda a, b: a+b, a = 1, b = 2)
>>> assert cell_go(c) == c(data = 3)
```

2.3.3 cell_item

pyg.base._cell.**cell_item**(*value*, *key=None*)

returns an item from a cell (if not cell, returns back the value). If no key is provided, will return the output of the cell

Parameters

value [cell or object or list of cells/objects] DESCRIPTION.

key [str, optional] The key within cell we are interested in. Note that key is treated as GUIDANCE only. Our strong preference is to return valid output from cell_output(cell)

Example non cells

```
>>> assert cell_item(1) == 1
>>> assert cell_item(dict(a=1,b=2)) == dict(a=1,b=2)
```

Example cells, simple

```
>>> c = cell(lambda a, b: a+b, a = 1, b = 2)
>>> assert cell_item(c) is None
>>> assert cell_item(c.go()) == 3
```

2.3.4 cell_func

pyg.base._cell.**cell_func**()

cell_func is a wrapped and wraps a function to act on cells rather than just on values

When called, it will returns not just the function, but also args, kwargs used to call it.

Example

```
>>> from pyg import *
>>> a = cell(lambda x: x**2, x  = 3)
>>> b = cell(lambda y: y**3, y  = 2)
>>> function = lambda a, b: a+b
>>> self = cell_func(function)
>>> result, args, kwargs = self(a,b)
```

```
>>> assert result == 8 + 9
>>> assert args[0].data == 3 ** 2
>>> assert args[1].data == 2 ** 3
```

2.4 encode and decode/save and load

2.4.1 encode

pyg.base._encode.**encode**(*value*)

encode/decode are performed prior to sending to mongodb or after retrieval from db. The idea is to make object embedding in Mongo transparent to the user.

- We use jsonpickle package to embed general objects. These are encoded as strings and can be decoded as long as the original library exists when decoding.
- pandas.DataFrame are encoded to bytes using pickle while numpy arrays are encoded using the faster array.tobytes() with arrays' shape & type exposed and searchable.

Example

```
>>> from pyg import *; import numpy as np
>>> value = Dict(a=1,b=2)
>>> assert encode(value) == {'a': 1, 'b': 2, '_obj': '{"py/type": "pyg.base._dict.
↪Dict"}'}
>>> assert decode({'a': 1, 'b': 2, '_obj': '{"py/type": "pyg.base._dict.Dict"}'})␣
↪== Dict(a = 1, b=2)
>>> value = dictable(a=[1,2,3], b = 4)
>>> assert encode(value) == {'a': [1, 2, 3], 'b': [4, 4, 4], '_obj': '{"py/type":
↪"pyg.base._dictable.dictable"}'}
>>> assert decode(encode(value)) == value
>>> assert encode(np.array([1,2])) ==  {'data': bytes,
>>>                                      'shape': (2,),
>>>                                      'dtype': '{"py/reduce": [{"py/type":
↪"numpy.dtype"}, {"py/tuple": ["i4", false, true]}, {"py/tuple": [3, "<", null,␣
↪null, null, -1, -1, 0]}]}',
>>>                                      '_obj': '{"py/function": "pyg.base._
↪encode.bson2np"}'}
```

Example functions and objects

```
>>> from pyg import *; import numpy as np
>>> assert encode(ewma) == '{"py/function": "pyg.timeseries._ewm.ewma"}'
>>> assert encode(Calendar) == '{"py/type": "pyg.base._drange.Calendar"}'
```

Parameters

value [obj] An object to be encoded

Returns

A pre-json object

2.4.2 decode

`pyg.base._encode.decode(value, date=None)`

decodes a string or an object dict

Parameters

value [str or dict] usually a json

date [None, bool or a regex expression, optional] date format to be decoded

Returns

obj the json decoded.

Examples

```
>>> from pyg import *
>>> class temp(dict):
>>>     pass
```

```
>>> orig = temp(a = 1, b = dt(0))
>>> encoded = encode(orig)
>>> assert eq(decode(encoded), orig) # type matching too...
```

2.4.3 pd_to_parquet

`pyg.base._parquet.pd_to_parquet(value, path, compression='GZIP')`

a small utility to save df to parquet, extending both pd.Series and non-string columns

Example

```
>>> from pyg import *
>>> import pandas as pd
>>> import pytest
```

```
>>> df = pd.DataFrame([[1,2],[3,4]], drange(-1), columns = [0, dt(0)])
>>> s = pd.Series([1,2,3], drange(-2))
```

```
>>> with pytest.raises(ValueError): ## must have string column names
        df.to_parquet('c:/temp/test.parquet')
```

```
>>> with pytest.raises(AttributeError): ## pd.Series has no to_parquet
        s.to_parquet('c:/temp/test.parquet')
```

```
>>> df_path = pd_to_parquet(df, 'c:/temp/df.parquet')
>>> series_path = pd_to_parquet(s, 'c:/temp/series.parquet')
```

```
>>> df2 = pd_read_parquet(df_path)
>>> s2 = pd_read_parquet(series_path)
```

```
>>> assert eq(df, df2)
>>> assert eq(s, s2)
```

2.4.4 pd_read_parquet

pyg.base._parquet.**pd_read_parquet**(*path*)

a small utility to read df/series from parquet, extending both pd.Series and non-string columns

Example

```
>>> from pyg import *
>>> import pandas as pd
>>> import pytest
```

```
>>> df = pd.DataFrame([[1,2],[3,4]], drange(-1), columns = [0, dt(0)])
>>> s = pd.Series([1,2,3], drange(-2))
```

```
>>> with pytest.raises(ValueError): ## must have string column names
        df.to_parquet('c:/temp/test.parquet')
```

```
>>> with pytest.raises(AttributeError): ## pd.Series has no to_parquet
        s.to_parquet('c:/temp/test.parquet')
```

```
>>> df_path = pd_to_parquet(df, 'c:/temp/df.parquet')
>>> series_path = pd_to_parquet(s, 'c:/temp/series.parquet')
```

```
>>> df2 = pd_read_parquet(df_path)
>>> s2 = pd_read_parquet(series_path)
```

```
>>> assert eq(df, df2)
>>> assert eq(s, s2)
```

2.4.5 parquet_encode

pyg.mongo._encoders.**parquet_encode**(*value, path, compression='GZIP'*)

encodes a single DataFrame or a document containing dataframes into a an abject that can be decoded

```
>>> from pyg import *
>>> path = 'c:/temp'
>>> value = dict(key = 'a', n = np.random.normal(0,1, 10), data = dictable(a =
↪[pd.Series([1,2,3]), pd.Series([4,5,6])], b = [1,2]), other = dict(df = pd.
↪DataFrame(dict(a=[1,2,3], b= [4,5,6]))))
>>> encoded = parquet_encode(value, path)
>>> assert encoded['n']['file'] == 'c:/temp/n.npy'
>>> assert encoded['data'].a[0]['path'] == 'c:/temp/data/a/0.parquet'
>>> assert encoded['other']['df']['path'] == 'c:/temp/other/df.parquet'
```

```
>>> decoded = decode(encoded)
>>> assert eq(decoded, value)
```

2.4.6 csv_encode

`pyg.mongo._encoders.csv_encode(value, path)`

encodes a single DataFrame or a document containing dataframes into a an object that can be decoded while saving dataframes into csv

```
>>> path = 'c:/temp'
>>> value = dict(key = 'a', data = dictable(a = [pd.Series([1,2,3]), pd.Series([4,
↪5,6])], b = [1,2]), other = dict(df = pd.DataFrame(dict(a=[1,2,3], b= [4,5,
↪6]))))
>>> encoded = csv_encode(value, path)
>>> assert encoded['data'].a[0]['path'] == 'c:/temp/data/a/0.csv'
>>> assert encoded['other']['df']['path'] == 'c:/temp/other/df.csv'
```

```
>>> decoded = decode(encoded)
>>> assert eq(decoded, value)
```

2.4.7 convertors to bytes

`pyg.base._encode.pd2bson(value)`

converts a value (usually a pandas.DataFrame/Series) to bytes using pickle

`pyg.base._encode.np2bson(value)`

converts a numpy array to bytes using value.tobytes(). This is much faster than pickle but does not save shape/type info which we save separately.

`pyg.base._encode.bson2np(data, dtype, shape)`

converts a byte with dtype and shape information into a numpy array.

`pyg.base._encode.bson2pd(data)`

converts a pickled object back to an object. We insist that new object has .shape to ensure we did not unpickle gibberish.

2.5 dates and calendar

2.5.1 dt

`pyg.base._dates.dt(*args, dialect='uk', none=<built-in method now of type object>)`

A more generic constructor for datetime.datetime.

Example Simple construction

```
>>> assert dt(2000,1 ,1) == datetime.datetime(2000, 1, 1, 0, 0) # name of month
>>> assert dt(2000,'jan',1) == datetime.datetime(2000, 1, 1, 0, 0) # name of month
>>> assert dt(2000,'f',1) == datetime.datetime(2000, 1, 1, 0, 0) # future month␣
↪code
>>> assert dt('01-02-2002') == datetime.datetime(2002, 2, 1)
>>> assert dt('01-02-2002', dialect = 'US') == datetime.datetime(2002, 1, 2)
>>> assert dt('01 March 2002') == datetime.datetime(2002, 3, 1)
>>> assert dt('01 March 2002', dialect = 'US') == datetime.datetime(2002, 3, 1)
>>> assert dt('01 March 2002 10:20:30') == datetime.datetime(2002, 3, 1, 10, 20,␣
↪30)
```

```
>>> assert dt(20020301) == datetime.datetime(2002, 3, 1)
>>> assert dt(37316) == datetime.datetime(2002, 3, 1) # excel date
>>> assert dt(730180) == datetime.datetime(2000,3,1) # ordinal for 1/3/2000
>>> assert dt(2000,3,1).timestamp() == 951868800.0
>>> assert dt(951868800.0) == datetime.datetime(2000,3,1) # utc timestamp
>>> assert dt(np.datetime64(dt(2000,3,1))) == dt(2000,3,1) ## numpy.datetime64␣
↪object
```

```
>>> assert dt(2000) == datetime.datetime(2000,1,1)
>>> assert dt(2000,3) == datetime.datetime(2000,3,1)
>>> assert dt(2000,3, 1) == datetime.datetime(2000,3,1)
>>> assert dt(2000,3, 1, 10,20,30) == datetime.datetime(2000,3,1,10,20,30)
>>> assert dt(2000,'march', 1) == datetime.datetime(2000,3,1)
>>> assert dt(2000,'h', 1) == datetime.datetime(2000,3,1) # future codes
```

Example date as offset from today

```
>>> today = dt(0);
>>> import datetime
>>> day = datetime.timedelta(1)
>>> assert dt(-3) == today - 3 * day
>>> assert dt('-10b') == today - 14 * day
```

Example datetime arithmetic:

dt has an interesting logic in implementing datetime arithmentic:

- day and month parameters can be negative or bigger than the days of month
- dt() will roll back/forward from the date which is valid

```
>>> assert dt(2000,4,1) == datetime.datetime(2000, 4, 1, 0, 0)
>>> assert dt(2000,4,0) == datetime.datetime(2000, 3, 31, 0, 0) # a day before␣
↪dt(2000,4,1)
```

and rolling back months:

```
>>> assert dt(2000,0,1) == datetime.datetime(1999, 12, 1, 0, 0) # a month before␣
↪dt(2000,1,1)
>>> assert dt(2000,13,1) == datetime.datetime(2001, 1, 1, 0, 0) # a month after␣
↪dt(2000,12,1)
```

This may feel unnatural at first, but does allow for much nicer code, e.g.: [dt(2000,i,1) for i in range(-10,10)]

Parameters

***args** [str, int or dates] argument to be converted into dates

dialect [str, optional] parsing of 01/02/2020 is it 1st Feb or 2nd Jan? The default is 'uk', i.e. dd/mm/yyyy

none [callable, optional] What is dt()? The default is datetime.datetime.now()

2.5.2 ymd

pyg.base._dates.**ymd**(**args, dialect='uk', none=<built-in method now of type object>*)

just like dt() but always returns date only (year/month/date) without fractions. see dt() for full documentation

datetime.datetime

2.5.3 dt_bump

pyg.base._dates.**dt_bump**(*t, *bumps*)

Example

```
>>> from pyg import *
>>> t  = pd.Series([1,2,3], drange(dt(2000,1,1),2))
>>> assert eq(dt_bump(t, 1), pd.Series([1,2,3], drange(dt(2000,1,2),2)))
```

2.5.4 drange

pyg.base._drange.**drange**(*t0=None, t1=None, bump=None*)

A quick and happy wrapper for dateutil.rrule

Examples

```
>>> drange(2000, 10, 1) # 10 days starting from dt(2000,1,1)
>>> drange(2000, '10b', '1b') # weekdays between dt(2000,1,1) and dt(2000,1,17)
>>> drange('-10b', 0, '1b') # business days since 10 bdays ago
>>> drange('-10b', '10b', '1w') # starting 10b days ago, to 10b from now,␣
↪counting in weekly jumps
```

Parameters

t0 [date, optional] start date. The default is None.

t1 [date, optional] end date. The default is None.

bump [timedelta, int, string, optional] bump period. The default is None.

Returns

list of dates

Example

```
>>> t0 = 2000; t1 = 1999
>>> bump = '-1b'
```

Example

```
>>> t0 = dt(2020); t1 = dt(2021); bump = datetime.timedelta(hours = 4)
```

2.5.5 date_range

pyg.base._drange.**date_range**(*t0=None*, *t1=None*)

2.5.6 Calendar

class pyg.base._drange.**Calendar**(*key=None*, *holidays=None*, *weekend=None*, *t0=None*, *t1=None*, *adj='m'*, *day_start=0*, *day_end=235959*)

Calendar is

- a dict
- containing holiday dates
- implementing business day arithmetic

Calendar is restricted to operate between cal.t0 and cal.t1 which default to TMIN = 1900 and TMAX = 2300

Calendar does this by having two key members:

- dt2int: a mapping from all business dates to their integer 'clock'
- int2dt: a mapping from integer value to the date

Since Calendar is an 'expensive' memory wise, we assign a key to the calendar and the Calendar is stored in the singleton calendars under this key

Example

```
>>> from pyg import *
>>> holidays = dictable([[1,'2012-01-02','New Year Day',],
                        [2,'2012-01-16','Martin Luther King Jr. Day',],
                        [3,'2012-02-20','Presidents Day (Washingtons Birthday)',],
                        [4,'2012-05-28','Memorial Day',],
                        [5,'2012-07-04','Independence Day',],
                        [6,'2012-09-03','Labor Day',],
                        [7,'2012-10-08','Columbus Day',],
                        [8,'2012-11-12','Veterans Day',],
                        [9,'2012-11-22','Thanksgiving Day',],
                        [10,'2012-12-25','Christmas Day',],
                        [11,'2013-01-01','New Year Day',],
                        [12,'2013-01-21','Martin Luther King Jr. Day',],
                        [13,'2013-02-18','Presidents Day (Washingtons Birthday)',
↪],
                        [14,'2013-05-27','Memorial Day',],
                        [15,'2013-07-04','Independence Day',],
                        [16,'2013-09-02','Labor Day',],
                        [17,'2013-10-14','Columbus Day',],
                        [18,'2013-11-11','Veterans Day',],
                        [19,'2013-11-28','Thanksgiving Day',],
                        [20,'2013-12-25','Christmas Day',],
                        [21,'2014-01-01','New Year Day',],
                        [22,'2014-01-20','Martin Luther King Jr. Day',],
                        [23,'2014-02-17','Presidents Day (Washingtons Birthday)',
↪],
                        [24,'2014-05-26','Memorial Day',],
                        [25,'2014-07-04','Independence Day',],
                        [26,'2014-09-01','Labor Day',],
                        [27,'2014-10-13','Columbus Day',],
```

(continues on next page)

(continued from previous page)

```
                            [28,'2014-11-11','Veterans Day',],
                            [29,'2014-11-27','Thanksgiving Day',],], ['i', 'date',
↪'name']).do(dt, 'date')
```

```
>>> cal = calendar('US', holidays.date, t0 = 2012, t1 = 2015)
>>> assert not cal.is_bday(dt(2013,9,2))         # Labor day
```

```
>>> cached_calendar = calendar('US')
>>> assert not cached_calendar.is_bday(dt(2013,9,2))    # Labor day
```

```
>>> assert cal.adjust(dt(2013,9,2)) == dt(2013,9,3)
>>> assert cal.drange(dt(2013,9,0), dt(2013,9,7), '1b') == [dt(2013,8,30),␣
↪dt(2013,9,3), dt(2013,9,4), dt(2013,9,5), dt(2013,9,6),] ## skipped labour day␣
↪and weekend prior
```

```
>>> assert cal.bdays(dt(2013,9,0), dt(2013,9,7)) == 5
```

adjust (*date*, *adj=None*)
adjust a non-business day to prev/following bussiness date

Parameters

date : datetime. adj : None or p/f/m

adjustment convention: 'prev/following/modified following'

Returns

dateime nearby business day

dt_bump (*t*, *bump*, *adj=None*)
adds a bump to a date

Parameters

t [datetime] date to bump.

bump [int, str] bump e.g. '-1y' or '1b' or 3

adj [adjustement type] The default is None.

Returns

datetime bumped date.

is_trading (*date=None*)
calculates if we are within a trading session

Parameters

date [datetime, optional] the time & date we want to check. The default is None (i.e. now)

Returns

bool: are we within a trading session

trade_date(*date=None*, *adj=None*)

This is very similar for adjust, but it also takes into account the time of the day. if day_start = 0 and day_end = 23:59:59 then this is exactly adjust.

Parameters

date [datetime, optional] date (with time). The default is None.

adj [f/p, optional] If date isn't within trading day, which direction to adjust to? The default is None.

Example

```
>>> from pyg import *; import datetime
```

```
>>> uk = calendar('UK', day_start = 8, day_end = 17)
>>> assert uk.trade_date(dt(2021,2,9,5), 'f') == dt(2021, 2, 9)  # Tuesday␣
↪morning rolls into Tuesday
>>> assert uk.trade_date(dt(2021,2,9,5), 'p') == dt(2021, 2, 8)  # Tuesday␣
↪morning back into Monday
>>> assert uk.trade_date(dt(2021,2,7,5), 'f') == dt(2021, 2, 8)  # Sunday␣
↪rolls into Monday
>>> assert uk.trade_date(dt(2021,2,7,5), 'p') == dt(2021, 2, 5)  # Sunday␣
↪rolls back to Friday
```

```
>>> assert uk.trade_date(date = dt(2021,2,9,23), adj = 'f') == dt(2021, 2,␣
↪10)  # Tuesday eve rolls into Wed
>>> assert uk.trade_date(date = dt(2021,2,9,23), adj = 'p') == dt(2021, 2, 9)␣
↪ # Tuesday eve back into Tuesday
>>> assert uk.trade_date(date = dt(2021,2,7,23), adj = 'f') == dt(2021, 2, 8)␣
↪ # Sunday rolls into Monday
>>> assert uk.trade_date(date = dt(2021,2,7,23), adj = 'p') == dt(2021, 2, 5)␣
↪ # Sunday rolls back to Friday
```

```
>>> assert uk.trade_date(date = dt(2021,2,9,12), adj = 'f') == dt(2021, 2, 9)␣
↪ # Tuesday is Tuesday
>>> assert uk.trade_date(date = dt(2021,2,9,12), adj = 'p') == dt(2021, 2, 9)␣
↪ # Tuesday is Tuesday
```

```
>>> au = calendar('AU', day_start = 2230, day_end = 1300)
>>> assert au.trade_date(dt(2021,2,9,5), 'f') == dt(2021, 2, 9)  # Tuesday␣
↪morning in session
>>> assert au.trade_date(dt(2021,2,9,5), 'p') == dt(2021, 2, 9)  # Tuesday␣
↪morning in session
>>> assert au.trade_date(dt(2021,2,7,5), 'f') == dt(2021, 2, 8)  # Sunday␣
↪rolls into Monday
>>> assert au.trade_date(dt(2021,2,7,5), 'p') == dt(2021, 2, 5)  # Sunday␣
↪rolls back to Friday
```

```
>>> assert au.trade_date(date = dt(2021,2,9,23), adj = 'f') == dt(2021, 2,␣
↪10)  # Tuesday eve rolls into Wed
>>> assert au.trade_date(date = dt(2021,2,9,23), adj = 'p') == dt(2021, 2,␣
↪10)  # Already in Wed
>>> assert au.trade_date(date = dt(2021,2,7,23), adj = 'f') == dt(2021, 2, 8)␣
↪ # Sunday rolls into Monday
>>> assert au.trade_date(date = dt(2021,2,7,23), adj = 'p') == dt(2021, 2, 8)␣
↪ # Already on Monday
```

(continues on next page)

(continued from previous page)

```
>>> assert au.trade_date(date = dt(2021,2,5,23), adj = 'f') == dt(2021, 2, 8)␣
↪ # Friday afternoon rolls into Monday
```

```
>>> assert au.trade_date(date = dt(2021,2,9,14), adj = 'f') == dt(2021, 2,␣
↪10)  # Tuesday is over, roll to Wed
>>> assert au.trade_date(date = dt(2021,2,9,14), adj = 'p') == dt(2021, 2, 9)␣
↪ # roll back to Tues
```

2.5.7 calendar

pyg.base._drange.**calendar**(*key=None*, *holidays=None*, *weekend=None*, *t0=None*, *t1=None*, *day_start=0*, *day_end=235959*)

A function to returns either an existing calendar or construct a new one. - calendar('US') will return a US calendar if that is already cached - calendar('US', us_holiday_dates) will construct a calendar with holiday dates and then cache it

2.5.8 as_time

pyg.base._drange.**as_time**(*t=None*)

parses t into a datetime.time object

Example

```
>>> assert as_time('10:30:40') == datetime.time(10, 30, 40)
>>> assert as_time('103040') == datetime.time(10, 30, 40)
>>> assert as_time('10:30') == datetime.time(10, 30)
>>> assert as_time('1030') == datetime.time(10, 30)
>>> assert as_time('05') == datetime.time(5)
>>> assert as_time(103040) == datetime.time(10, 30, 40)
>>> assert as_time(13040) == datetime.time(1, 30, 40)
>>> assert as_time(130) == datetime.time(1, 30)
>>> assert as_time(datetime.time(1, 30)) == datetime.time(1, 30)
>>> assert as_time(datetime.datetime(2000, 1, 1, 1, 30)) == datetime.time(1, 30)
```

t [str/int/datetime.time/datetime.datetime] time of day

datetime.time

2.5.9 clock

pyg.base._drange.**clock**(*ts*, *time=None*, *t=None*)

returns a vector marking the passage of time.

Parameters

ts : timeseries time : None, a string or a Calendar, or already a timeseries of times

None: Will increment by 1 every non-nan observation 'i' : increment by 1 every date in index (nan or not) 'b' : weekdays distance 'd' : day-distance (ignore intraday stamp) 'f' : fraction-of-day-distance (do not ignore intraday stamp) 'm' : month-distance 'q' : quarter-distance 'y' : year-distance calendar: uses the business-days distance between any two dates

t: starting time in the past.

Returns

an array an increasing array of time such that distance between points match the above.

Example

```
>>> from pyg import *
>>> assert eq(clock(pd.Series(np.arange(10), drange(2000, 9))), np.arange(1,11))
>>> assert eq(clock(pd.Series(np.arange(10), drange(2000, 9)), t = 5), np.
↪arange(6,16))
>>> assert eq(clock(pd.Series(np.arange(10), drange(2000, 9)), 'i'), np.arange(1,
↪11))
>>> assert eq(clock(pd.Series(np.arange(10), drange(2000, 9)), 'b'), np.
↪array([26090, 26090, 26090, 26091, 26092, 26093, 26094, 26095, 26095, 26095]))
>>> assert eq(clock(pd.Series(np.arange(10), drange(2000, '9b', '1b')), 'b'), np.
↪arange(26090, 26100))
```

```
>>> assert eq(clock(np.arange(10)), np.arange(1,11))
>>> assert eq(clock(pd.Series(np.arange(10)), t = 5), np.arange(6,16))
>>> assert eq(clock(np.arange(10), 'i'), np.arange(1,11))
```

2.6 text manipulation

2.6.1 lower

pyg.base._txt.**lower**(*value*)

equivalent to txt.lower() but:

- does not throw on non-string
- supports lists/dicts

Example

```
>>> assert lower(['The Brown Fox',1]) == ['the brown fox',1]
>>> assert lower(dict(a = 'The Brown Fox', b = 3.0)) ==  {'a': 'the brown fox', 'b
↪': 3.0}
```

2.6.2 upper

pyg.base._txt.**upper**(*value*)

equivalent to txt.upper() but:

- does not throw on non-string
- supports lists/dicts

Example

```
>>> assert upper(['The Brown Fox',1]) == ['THE BROWN FOX',1]
>>> assert upper(dict(a = 'The Brown Fox', b = 3.0)) ==  {'a': 'THE BROWN FOX', 'b
↪': 3.0}
```

(continues on next page)

(continued from previous page)

2.6.3 proper

pyg.base._txt.**proper**(*value*)

equivalent to Excel's PROPER(txt) but:

- does not throw on non-string
- supports lists/dicts

Example

```
>>> assert proper(['THE BROWN FOX',1]) == ['The Brown Fox',1]
>>> assert proper(dict(a = 'THE BROWN FOX', b = 3.0)) ==  {'a': 'The Brown Fox',
↪'b': 3.0}
```

2.6.4 capitalize

pyg.base._txt.**capitalize**(*value*)

equivalent to text.capitalize() but:

- does not throw on non-string
- supports lists/dicts

Example

```
>>> assert capitalize('alan howard') == 'Alan howard' # use proper to get Alan␣
↪Howard
>>> assert capitalize(['alan howard', 'donald trump']) == ['Alan howard', 'Donald␣
↪trump'] # use proper?
```

2.6.5 strip

pyg.base._txt.**strip**(*value*)

equivalent to txt.strip() but:

- does not throw on non-string
- supports lists/dicts

Example

```
>>> assert strip([' whatever you say  ',' whatever you do..   ']) == ['whatever␣
↪you say', 'whatever you do..']
>>> assert strip(dict(a = ' whatever you say  ', b = 3.0)) ==  {'a': 'whatever␣
↪you say', 'b': 3.0}
```

2.6.6 split

`pyg.base._txt.split` (*text*, *sep=' '*, *dedup=False*)

equivalent to txt.split(sep) but supporsts:

- does not throw on non-string
- removal of multiple seps
- ensuring there is a unique single separator

Parameters

text [str] text to be stipped.

sep [str, list of str, optional] text used to strip. The default is ' '.

dedup [bool, optional] If True, will remove duplicated instances of seps. The default is False.

Returns

str splitted text

Example

```
>>> text = '   The quick... brown .. fox... '
>>> assert split(text) == ['', '', '', 'The', 'quick...', 'brown', '..', 'fox...',
↪ '']
>>> assert split(text, [' ', '.'], True) == ['The', 'quick', 'brown', 'fox']
>>> text = dict(a = 'Can split this', b = '..and split this too')
>>> assert split(text, [' ', '.'], True) == {'a': ['Can', 'split', 'this'], 'b': [
↪'and', 'split', 'this', 'too']}
```

2.6.7 replace

`pyg.base._txt.replace` (*text*, *old*, *new=None*)

A souped up version of text.replace(old, new)

Example replace continues to replace until no-more is found

```
>>> assert replace('this    has lots  of    double    spaces', ' '*2, ' ') ==
↪'this has lots of double spaces'
>>> assert replace('this, sentence? has! too, many, punctuations!', list(',?!.'))␣
↪== 'this sentence has too many punctuations'
>>> assert replace(dict(a = 1, b = [' text within a list ', 'and within a dict']),
↪ ' ') == {'a': 1, 'b': ['textwithinalist', 'andwithinadict']}
```

2.6.8 common_prefix

pyg.base._txt.**common_prefix**(**values*)

Parameters

***values** [list of iterables] values for which we want to find common prefix

Returns

iterable the common prefix.

Example

```
>>> assert common_prefix(['abra', 'abba', 'abacus']) == 'ab'
>>> assert common_prefix('abra', 'abba', 'abacus') == 'ab'
>>> assert common_prefix() is None
>>> assert common_prefix([1,2,3,4], [1,2,3,5,8]) == [1,2,3]
```

2.7 files & directory

2.7.1 mkdir

pyg.base._file.**mkdir**(*path*)
makes a new directory if not exists. It works if path is a filename too.

2.7.2 read_csv

pyg.base._file.**read_csv**(*path*)
light-weight csv reader, unlike pandas heavy duty :-)

2.8 tree manipulation

Trees are dicts of dicts. just like an item in a dict is (key, value), tree items are just longer tuples: (key1, key2, key3, value)

2.8.1 tree_to_items

pyg.base._dict.**tree_to_items**(*tree*, *types=(<class 'dict'>, <class 'pyg.base._dict.Dict'>, <class 'pyg.base._dictattr.dictattr'>)*)
An extension of dict.items(), returning a list of tuples but of varying length, each a branch of a tree

Parameters

tree [dict of dicts] a tree of data.

types [dict or a list of dict-types, optional] The types that we consider as 'branches' of the tree. Default is (dict, Dict, dictattr).

Returns

a list of tuples these are an extension of dict.items() and are of varying length

Example

```
>>> school = dict(pupils = dict(id1 = dict(name = 'james', surname = 'maxwell',␣
↪gender = 'm'),
                              id2 = dict(name = 'adam', surname = 'smith', gender = 'm'),
                              id3 = dict(name = 'michell', surname = 'obama', gender = 'f
↪'),
                              ),
        teachers = dict(math = dict(name = 'albert', surname = 'einstein', grade␣
↪= 3),
                                english = dict(name = 'william', surname = 'shakespeare',␣
↪grade = 3),
                                physics = dict(name = 'richard', surname = 'feyman',␣
↪grade = 4)
                                ))
```

```
>>> items = tree_to_items(school)
>>> items
```

[('pupils', 'id1', 'name', 'james'), ('pupils', 'id1', 'surname', 'maxwell'), ('pupils', 'id1', 'gender', 'm'), ('pupils', 'id2', 'name', 'adam'), ('pupils', 'id2', 'surname', 'smith'), ('pupils', 'id2', 'gender', 'm'), ('pupils', 'id3', 'name', 'michell'), ('pupils', 'id3', 'surname', 'obama'), ('pupils', 'id3', 'gender', 'f'), ('teachers', 'math', 'name', 'albert'), ('teachers', 'math', 'surname', 'einstein'), ('teachers', 'math', 'grade', 3), ('teachers', 'english', 'name', 'william'), ('teachers', 'english', 'surname', 'shakespeare'), ('teachers', 'english', 'grade', 3), ('teachers', 'physics', 'name', 'richard'), ('teachers', 'physics', 'surname', 'feyman'), ('teachers', 'physics', 'grade', 4)]

#To reverse this, we call:

```
>>> assert items_to_tree(items) == school
```

2.8.2 items_to_tree

pyg.base._dict.**items_to_tree**(*items, tree=None, raise_if_duplicate=True, ignore=None, types=(<class 'dict'>, <class 'pyg.base._dict.Dict'>, <class 'pyg.base._dictattr.dictattr'>)*)

converts **items** to branches of a tree. If an original **tree** is provided, hang the additional branches on the existing tree If **ignore** is provided as a list of values, will not overwrite branches with last value (the leaf) in these values

Example

```
>>> items = [('cambridge', 'smith', 'economics',),
         ('cambridge', 'keynes', 'economics'),
         ('cambridge', 'lyons',  'maths'),
         ('cambridge', 'maxwell', 'maths'),
         ('oxford', 'penrose', 'maths'),
         ]
```

```
>>> tree = items_to_tree(items)
>>> print(tree_repr(tree))
```

```
>>> cambridge:
>>>     smith:
>>>         economics
>>>     keynes:
>>>         economics
>>>     lyons:
>>>         maths
>>>     maxwell:
>>>         maths
>>> oxford:
>>>     {'penrose': 'maths'}
```

We can add to tree:

Parameters

items [list of tuples,] items are just like dict items, only longer,

tree [tree, optional] a pre-existing tree of trees. The default is None.

raise_if_duplicate [TYPE, optional] DESCRIPTION. The default is True.

ignore [list, optional] list of values that when over-writing an existing tree, should ignore. The default is None.

Example using ignore

```
>>> tree = dict(a = 1, b = 'keep_old_value')
>>> update = dict(a = 'valid_new_value', b = None, c = None)
>>> tree_update(tree, update, ignore = [None])
>>> {'a': valid_new_value, 'b': 'keep_old_value', 'c': None}
```

- a is over-ridden as the new value is valid
- b is not over-ridden since the update b = None is considereed invalid
- c is added as it did not exist before, even though c = None is invalid value

Returns

tree : dict of dicts

2.8.3 tree_update

`pyg.base._dict.`**`tree_update`**(*tree*, *update*, *types=(<class 'dict'>, <class 'pyg.base._dict.Dict'>, <class 'pyg.base._dictattr.dictattr'>)*, *ignore=None*)

equivalent to dict.update() except: not in-place and also updates further down the tree

Example

```
>>> ranking = dict(cambridge = dict(trinity = 1, stjohns = 2, christ = 3),
        oxford = dict(trinity = 1, jesus = 2, magdalene = 3))
>>> new_ranking = dict(oxford = dict(wolfson = 3, magdalene = 4))
```

```
>>> print(tree_repr(tree_update(ranking, new_ranking)))
```

cambridge: {‘trinity’: 1, ‘stjohns’: 2, ‘christ’: 3}

oxford: {'trinity': 1, 'jesus': 2, 'magdalene': 4, 'wolfson': 3}

Note how values for magdalene in Oxford were overwritten even though they are further down the tree

Example using ignore

```
>>> update = dict(a = None, b = np.nan, c = 0)
>>> tree = dict(a = 1, b = 2, c = 3)
>>> assert tree_update(tree, update) == update
>>> assert tree_update(tree, update, ignore = [None]) == dict(a = 1, b = np.nan,␣
↪c = 0)
>>> assert tree_update(tree, update, ignore = [None, np.nan]) == dict(a = 1, b =␣
↪2, c = 0)
>>> assert tree_update(tree, update, ignore = [None, np.nan, 0]) == tree
```

Parameters

tree [tree] existing tree.

update [tree] new information.

types [types, optional] see tree_to_items. The default is (dict, Dict, dictattr).

Returns

tree updated tree.

2.8.4 tree_to_table

pyg.base._tree.**tree_to_table**(*tree, pattern*)

The best way to understand is to give an example:

Examples

```
>>> school = dict(pupils = dict(id1 = dict(name = 'james', surname = 'maxwell',␣
↪gender = 'm'),
                          id2 = dict(name = 'adam', surname = 'smith', gender = 'm
↪'),
                          id3 = dict(name = 'michell', surname = 'obama', gender␣
↪= 'f'),
                          ),
              teachers = dict(math = dict(name = 'albert', surname = 'einstein',␣
↪grade = 3),
                            english = dict(name = 'william', surname =
↪'shakespeare', grade = 3),
                            physics = dict(name = 'richard', surname = 'feyman',␣
↪grade = 4)
                            ))
```

Suppose we wanted to identify all male students:

```
>>> res = tree_to_table(school, 'pupils/%id/gender/m')
>>> assert res == [dict(id = 'id1'), dict(id = 'id2')]
```

or grades: >>> res = tree_to_table(school, 'teachers/%subject/grade/%grade') >>> assert res == [{'grade': 3, 'subject': 'math'},

{'grade': 3, 'subject': 'english'}, {'grade': 4, 'subject': 'physics'}]

Parameters

tree [tree (dict of dicts)] tree is a yaml-like structure

pattern [string] The pattern whose instances we wish to find in tree

Returns

list of dicts

2.8.5 tree_repr

pyg.base._tree_repr.**tree_repr**(*value*, *offset=0*)

a cleaner representation of a tree

Example

```
>>> school = dict(pupils = dict(id1 = dict(name = 'james', surname = 'maxwell',␣
↪gender = 'm'),
                    id2 = dict(name = 'adam', surname = 'smith', gender = 'm'),
                    id3 = dict(name = 'michell', surname = 'obama', gender = 'f'),
                    ),
    teachers = dict(math = dict(name = 'albert', surname = 'einstein', grade = 3),
                    english = dict(name = 'william', surname = 'shakespeare',␣
↪grade = 3),
                    physics = dict(name = 'richard', surname = 'feyman', grade =␣
↪4)
                    ))
```

```
>>> print(tree_repr(school, 4))
pupils:
    id1:
        {'name': 'james', 'surname': 'maxwell', 'gender': 'm'}
    id2:
        {'name': 'adam', 'surname': 'smith', 'gender': 'm'}
    id3:
        {'name': 'michell', 'surname': 'obama', 'gender': 'f'}
teachers:
    math:
        {'name': 'albert', 'surname': 'einstein', 'grade': 3}
    english:
        {'name': 'william', 'surname': 'shakespeare', 'grade': 3}
    physics:
        {'name': 'richard', 'surname': 'feyman', 'grade': 4}
```

Parameters

value : a tree

offset [int, optional] offset from the left for printing. The default is 0.

Returns

TYPE DESCRIPTION.

2.9 list functions

2.9.1 as_list

pyg.base._as_list.**as_list**(*value*, *none=False*)

returns a list of the original object.

Example

```
>>> assert as_list(None) == []
>>> assert as_list(4) == [4]
>>> assert as_list((1,2,)) == [1,2]
>>> assert as_list([1,2,]) == [1,2]
>>> assert eq(as_list(np.array([1,2,])) , [np.array([1,2,])])
>>> assert as_list(dict(a = 1)) == [dict(a=1)]
```

In practice, this function is has an incredible useful usage:

Example using as_list to give flexibility on *args

```
>>> def my_sum(*values):
>>>     values = as_list(values)
>>>     return sum(values)
```

```
>>> assert my_sum(1,2,3) == 6
>>> assert my_sum([1,2,3]) == 6 ## This is nice... wasn't possible before
```

Parameters

value : anything none : bool optional

Shall I return None as a value? The default is False and we return [], if True, returns [None]

Returns

list a list of original objects.

2.9.2 as_tuple

pyg.base._as_list.**as_tuple**(*value*, *none=False*)

returns a tuple of the original object.

Example

```
>>> assert as_tuple(None) == ()
>>> assert as_tuple(4) == (4,)
>>> assert as_tuple((1,2,)) == (1,2)
>>> assert as_tuple([1,2,]) == (1,2)
>>> assert eq(as_tuple(np.array([1,2,])) , (np.array([1,2,]),))
>>> assert as_tuple(dict(a = 1)) == (dict(a=1),)
```

In practice, this function is has an incredible useful usage:

Example using as_list to give flexibility on *args

```
>>> def my_sum(*values):
>>>     values = as_tuple(values)
>>>     return sum(values)
```

```
>>> assert my_sum(1,2,3) == 6
>>> assert my_sum([1,2,3]) == 6 ## This is nice... wasn't possible before
```

Parameters

value : anything none : bool optional

Shall I return None as a value? The default is False and we return [], if True, returns [None]

Returns

tuple a tuple of original objects.

2.9.3 first

`pyg.base._as_list.first(value)`

returns the first value in a list (None if empty list) or the original if value not a list

Example

```
>>> assert first(5) == 5
>>> assert first([5,5]) == 5
>>> assert first([]) is None
>>> assert first([1,2]) == 1
```

2.9.4 last

`pyg.base._as_list.last(value)`

returns the last value in a list (None if empty list) or the original if value not a list

Example

```
>>> assert last(5) == 5
>>> assert last([5,5]) == 5
>>> assert last([]) is None
>>> assert last([1,2]) == 2
```

2.9.5 unique

`pyg.base._as_list.unique(value)`

returns the asserted unique value in a list (None if empty list) or the original if value not a list. Throws an exception if list non-unique

Example

```
>>> assert unique(5) == 5
>>> assert unique([5,5]) == 5
>>> assert unique([]) is None
>>> with pytest.raises(ValueError):
>>>     unique([1,2])
```

2.10 Comparing and Sorting

2.10.1 cmp

pyg.base._sort.**cmp**(*x*, *y*)

Implements lexcompare while allowing for comparison of different types. First compares by type, then by length, then by keys and finally on values

Parameters

x [obj] 1st object to be compared.

y [obj] 2nd object to be compared.

Returns

int returns -1 if x<y else 1 if x>y else 0

Examples

```
>>> assert cmp('2', 2) == 1
>>> assert cmp(np.int64(2), 2) == 0
>>> assert cmp(None, 2.0) == -1 # None is smallest
>>> assert cmp([1,2,3], [4,5]) == 1 # [1,2,3] is longer
>>> assert cmp([1,2,3], [1,2,0]) == 1 # lexical sorting
>>> assert cmp(dict(a = 1, b = 2), dict(a = 1, c = 2)) == -1 # lexical sorting on keys
>>> assert cmp(dict(a = 1, b = 2), dict(b = 2, a = 1)) == 0 # order does not matter
```

2.10.2 Cmp

pyg.base._sort.**Cmp**(*x*)

class wrapper of cmp, allowing us to compare objects of different types

Example

```
>>> with pytest.raises(TypeError):
>>>     sorted([1,2,3,None])
```

```
>>> # but this is fine:
>>> assert sorted([1,3,2,None], key = Cmp) == [None, 1, 2, 3]
```

2.10.3 sort

`pyg.base._sort.sort(iterable)`
implements sorting allowing for comparing of not-same-type objects

Parameters

iterable [iterable] values to be sorted

Returns

list sorted list.

Example

```
>>> with pytest.raises(TypeError):
>>>     sorted([1,2,3,None])
>>> # but this is fine:
>>> sort([1,3,2,None]) == [None, 1, 2, 3]
```

2.10.4 eq

`pyg.base._eq.eq(x, y)`
A better nan-handling equality comparison. Here is the problem:

```
>>> import numpy as np
>>> assert not np.nan == np.nan  ## What?
```

The nan issue extends to np.arrays. . .

```
>>> assert list(np.array([np.nan,2]) == np.array([np.nan,2])) == [False, True]
```

but not to lists. . .

```
>>> assert [np.nan] == [np.nan]
```

But wait, if the lists are derived from np.arrays, then no equality. . .

```
>>> assert not list(np.array([np.nan])) == list(np.array([np.nan]))
```

The other issue is inheritance:

```
>>> class FunnyDict(dict):
>>>     def __getitem__(self, key):
>>>         return 5
>>> assert dict(a = 1) == FunnyDict(a=1) ## equality seems to ignore any type␣
↪mismatch
>>> assert not dict(a = 1)['a'] == FunnyDict(a = 1)['a']
```

There are also issues with partial

```
>>> from functools import partial
>>> f = lambda a: a + 1
>>> x = partial(f, a = 1)
```

(continues on next page)

(continued from previous page)

```
>>> y = partial(f, a = 1)
>>> assert not x == y
```

```
>>> import pandas as pd
>>> import pytest
>>> from pyg import eq
```

```
>>> assert eq(np.nan, np.nan) ## That's better
>>> assert eq(x = np.array([np.nan,2]), y = np.array([np.nan,2]))
>>> assert eq(np.array([np.array([1,2]),2], dtype = 'object'), np.array([np.
↪array([1,2]),2], dtype = 'object'))
>>> assert eq(np.array([np.nan,2]),np.array([np.nan,2]))
>>> assert eq(dict(a = np.array([np.array([1,2]),2], dtype = 'object')) ,  dict(a␣
↪= np.array([np.array([1,2]),2], dtype = 'object')))
>>> assert eq(dict(a = np.array([np.array([1,np.nan]),np.nan])) ,  dict(a = np.
↪array([np.array([1,np.nan]),np.nan])))
>>> assert eq(np.array([np.array([1,2]),dict(a = np.array([np.array([1,2]),2]))]),
↪ np.array([np.array([1,2]),dict(a = np.array([np.array([1,2]),2]))]))
```

```
>>> assert not eq(dict(a = 1), FunnyDict(a=1))
>>> assert eq(1, 1.0)
>>> assert eq(x = pd.DataFrame([1,2]), y = pd.DataFrame([1,2]))
>>> assert eq(pd.DataFrame([np.nan,2]), pd.DataFrame([np.nan,2]))
>>> assert eq(pd.DataFrame([1,np.nan], columns = ['a']), pd.DataFrame([1,np.nan],␣
↪columns = ['a']))
>>> assert not eq(pd.DataFrame([1,np.nan], columns = ['a']), pd.DataFrame([1,np.
↪nan], columns = ['b']))
```

2.10.5 in

`pyg.base._eq.in_(x, seq)`
Evaluates if x is in seq, avoiding issues such as these:

```
>>> s = pd.Series([1,2,3])
>>> with pytest.raises(ValueError):
>>>     s in [None]
>>> assert not in_(s, [None])
>>> assert in_(s, [None, s])
```

2.11 bits and pieces

2.11.1 type functions

`pyg.base._types.is_arr(value)`
is value a numpy array of non-zero-size

`pyg.base._types.is_bool(value)`
is value a Bool, or a **np.bool_** type

`pyg.base._types.is_date(value)`
is value a date type: either datetime.date, datetime.datetime or np.datetime64

`pyg.base._types.is_df(value)`
: is value a pd.DataFrame

`pyg.base._types.is_dict(value)`
: is value a dict

`pyg.base._types.is_float(value)`
: is value an float, or any variant of np.float

`pyg.base._types.is_int(value)`
: is value an int, or any variant of np.intN type

`pyg.base._types.is_iterable(value)`
: is value Iterable excluding a string

`pyg.base._types.is_len(value)`
: is value of zero length (or has no len at all)

`pyg.base._types.is_list(value)`
: is value a list

`pyg.base._types.is_nan(value)`
: is value a nan or an inf. Unlike np.isnan, works for non numeric

`pyg.base._types.is_none(value)`
: is value None

`pyg.base._types.is_num(value)`
: is _int(value) or is_float(value)

`pyg.base._types.is_pd(value)`
: is value a pd.DataFrame/pd.Series

`pyg.base._types.is_series(value)`
: is value a pd.Series

`pyg.base._types.is_str(value)`
: is value a str, or a **np.str_** type

`pyg.base._types.is_ts(value)`
: is value a pandas datafrome whch is indexed by datetimes

`pyg.base._types.is_tuple(value)`
: is value a tuple

`pyg.base._types.nan2none(value)`
: convert np.nan/np.inf to None

2.11.2 zipper

`pyg.base._zip.zipper(*values)`
: a safer version of zip

Examples zipper works with single values as well as full list:

```
>>> assert list(zipper([1,2,3], 4)) == [(1, 4), (2, 4), (3, 4)]
>>> assert list(zipper([1,2,3], [4,5,6])) == [(1, 4), (2, 5), (3, 6)]
>>> assert list(zipper([1,2,3], [4,5,6], [7])) ==  [(1, 4, 7), (2, 5, 7), (3, 6,
↪7)]
```

(continues on next page)

(continued from previous page)

```
>>> assert list(zipper([1,2,3], [4,5,6], None)) ==  [(1, 4, None), (2, 5, None),␣
↪(3, 6, None)]
>>> assert list(zipper((1,2,3), np.array([4,5,6]), None)) ==  [(1, 4, None), (2,␣
↪5, None), (3, 6, None)]
```

Examples zipper rejects multi-length lists

```
>>> import pytest
>>> with pytest.raises(ValueError):
>>>     zipper([1,2,3], [4,5])
```

Parameters

***values** [lists] values to be zipped

Returns

zipped values

2.11.3 reducer

pyg.base._reducer.**reducer**(*function*, *sequence*, *default=None*)

reduce adds stuff to zero by defaults. This is not needed.

Parameters

function [callable] binary function.

sequence [iterable] list of inputs to be applied iteratively to reduce.

default [TYPE, optional] A default value to be returned with an empty sequence

Example

```
>>> from operator import add, mul
>>> from functools import reduce
>>> import pytest
```

```
>>> assert reducer(add, [1,2,3,4]) == 10
>>> assert reducer(mul, [1,2,3,4]) == 24
>>> assert reducer(add, [1]) == 1
```

```
>>> assert reducer(add, []) is None
>>> with pytest.raises(TypeError):
>>>     reduce(add, [])
```

2.11.4 reducing

class pyg.base._reducer.**reducing**(*function=None*, **args*, ***kwargs*)

Makes a bivariate-function being able to act on a sequence of elements using reduction

Example

```
>>> from operator import mul
>>> assert reducing(mul)([1,2,3,4]) == 24
>>> assert reducing(mul)(6,4) == 24
```

Since a.join(b).join(c).join(d) is also very common, we provide a simple interface for that:

Example chaining

```
>>> assert reducing('__add__')([1,2,3,4]) == 10
>>> assert reducing('__add__')(6,4) == 10
```

d = dictable(a = [1,2,3,5,4]) reducing('inc')(d, dict(a=1))

2.11.5 logger and get_logger

pyg.base._logger.**get_logger**(*name='pyg'*, *level='info'*, *fmt='%(asctime)s - %(name)s - %(levelname)s - %(message)s'*, *file=False*, *console=True*)

quick utility to simplify loggers creation and ensure we cache them and do not add to many handlers

Parameters

name [str, optional] name of logger. The default is 'pyg'.

level [str, optional] DEBUG/INFO/WARN etc. The default is 'info'.

fmt [str, optional] string formatting for messages. The default is '%(asctime)s - %(name)s - %(levelname)s - %(message)s'.

file [bool/str, optional] the name of the file to log to. The default is False = do not log to file.

console [bool, optional] log to console? The default is True.

Returns

logging.logger

2.11.6 access functions

These are useful to convert object-oriented code to declarative functions

pyg.base._getitem.**callattr**(*value*, *attr*, *args=None*, *kwargs=None*)

gets the attribute(s) from a value and calls its

Example

```
>>> from pyg import *
>>> value = Dict(function = lambda a, b: a + b)
>>> assert callattr(value, 'function', kwargs = dict(a = 1, b = 2)) == 3
>>> assert callattr(value, attr = 'function', args = (1, 2), kwargs = None) == 3
```

```
>>> ts = pd.Series(np.random.normal(0,1,1000))
>>> assert ts.std() == callattr(ts, 'std')
>>> assert eq(ts.ewm(com = 10).mean(), callattr(ts, ['ewm','mean'], kwargs = [{
↪'com':10}, {}]))
```

```
>>> d = dictable(a = [1,2,3,4,1,2], b = list('abcdef'))
>>> assert callattr(d, ['inc', 'exc'], kwargs = [dict(a = 2), dict(b = 'f')]) ==␣
↪d.inc(a = 2).exc(b = 'f')
```

value [obj] object that contrains an item.

attr [string(s)] key within object.

args [tuple, optional] tuple of values to be fed to function. The default is None.

kwargs [dict, optional] kwargs to be fed to the method. The default is None.

pyg.base._getitem.**callitem**(*value*, *key*, *args=None*, *kwargs=None*)

gets an item and calls it

Example

```
>>> c = dict(function = lambda a, b: a + b)
>>> assert callitem(c, 'function', kwargs = dict(a = 1, b = 2)) == 3
>>> assert callitem(c, 'function', args = (1, 2)) == 3
```

value [obj] object that contrains an item.

key [string] key within object.

args [tuple, optional] tuple of values to be fed to function. The default is None.

kwargs [dict, optional] kwargs to be fed to the method. The default is None.

pyg.base._getitem.**getitem**(*value*, *key*, **default*)

gets an item, like getattr

Example

```
>>> a = dict(a = 1)
>>> assert getitem(a, 'a') == 1
>>> assert getitem(a, 'b', 2) == 2
```

```
>>> import pytest
>>> with pytest.raises(KeyError):
>>>     getitem(a, 'b')
```

2.11.7 inspection

There are a few functions extending the inspect module.

pyg.base._inspect.**argspec_add**(*fullargspec*, ***update*)

adds new args with default values at the end of the existing args

Parameters

fullargspec [FullArgSpec] DESCRIPTION.

****update** [dict] parameter names with their default values.

Returns

FullArgSpec

Example

```
>>> f = lambda b : b
>>> argspec = getargspec(f)
>>> updated = argspec_add(argspec, axis = 0)
>>> assert updated.args == ['b', 'axis'] and updated.defaults == (0,)
```

```
>>> f = lambda b, axis : None ## axis already exists without a default
>>> argspec = getargspec(f)
>>> updated = argspec_add(argspec, axis = 0)
>>> assert updated == argspec
```

```
>>> f = lambda b, axis =1 : None ## axis already exists with a different default
>>> argspec = getargspec(f)
>>> updated = argspec_add(argspec, axis = 0)
>>> assert updated == argspec
```

pyg.base._inspect.**argspec_defaults**(*function*)

Returns the function defaults as a dict rather than using the inspect structure

Example

```
>>> f = lambda a, b = 1: a+b
>>> assert argspec_defaults(f) == dict(b=1)
```

```
>>> g = partial(f, b = 2)
>>> assert argspec_defaults(g) == dict(b=2)
```

Parameters

function : callable

Returns

defaults as a dict.

pyg.base._inspect.**argspec_required**(*function*)

Parameters

function : callable

Returns

list parameters that *must* be provided in order to run the function

pyg.base._inspect.**argspec_update**(*argspec*, ***kwargs*)

generic function to create new FullArgSpec (python 3) or normal ArgSpec (python 2)

Parameters

argspec [FullArgSpec] The argspec of the dunction

****kwargs** [TYPE] updates

Returns

FullArgSpec

Example

```
>>> f = lambda a, b =1 : a + b
>>> argspec = getargspec(f)
>>> assert argspec_update(argspec, args = ['a', 'b', 'c']) == inspect.
↪FullArgSpec(**{'annotations': {},
                                                                            'args
↪': ['a', 'b', 'c'],

↪'defaults': (1,),

↪'kwonlyargs': [],

↪'kwonlydefaults': None,

↪'varargs': None,

↪'varkw': None})
```

pyg.base._inspect.**call_with_callargs**(*function*, *callargs*)

replicates inspect.getcallargs with support to functions within decorators

Example

```
>>> function = lambda a, b, *args, **kwargs: 1+b+len(args)+10*len(kwargs)
>>> args = (1,2,3,4,5); kwargs = dict(c = 6, d = 7)
>>> assert function(*args, **kwargs) == 26
>>> callargs = getcallargs(function, *args, **kwargs)
>>> assert call_with_callargs(function, callargs) == 26
```

pyg.base._inspect.**getargs**(*function*, *n=0*)

Parameters

function [callable] The function for which we want the args

n [int optional] get the name opf the args after allowing for n args to be set by *args. The default is 0.

Returns

None or a list of args

pyg.base._inspect.**getargspec**(*function*)

Extends inspect.getfullargspec to allow us to decorate functions with a signature.

Parameters

function [callable] function for which we want to know argspec.

Returns

inspect.FullArgSpec

pyg.base._inspect.**getcallargs**(*function*, **args*, ***kwargs*)

replicates inspect.getcallargs with support to functions within decorators

Example

```
>>> from pyg import *; import inspect
>>> function = lambda a, b, *myargs, **mykwargs: 1
>>> args = (1,2,3,4,5); kwargs = dict(c = 6, d = 7)
>>> assert getcallargs(function, *args, **kwargs) == inspect.getcallargs(function,
↪ *args, **kwargs) == {'a': 1, 'b': 2, 'myargs': (3, 4, 5), 'mykwargs': {'c': 6,
↪'d': 7}}
```

```
>>> function = lambda a: a + 1
>>> args = (); kwargs = dict(a=1)
>>> assert getcallargs(function, *args, **kwargs) == inspect.getcallargs(function,
↪ *args, **kwargs) == dict(a = 1)
```

```
>>> function = lambda a, b = 1: 1
>>> args = (); kwargs = dict(a=1)
>>> assert getcallargs(function, *args, **kwargs) == inspect.getcallargs(function,
↪ *args, **kwargs) == dict(a = 1, b = 1)
>>> args = (); kwargs = dict(a=1, b = 2)
>>> assert getcallargs(function, *args, **kwargs) == inspect.getcallargs(function,
↪ *args, **kwargs) == dict(a = 1, b = 2)
>>> args = (1,); kwargs = {}
>>> assert getcallargs(function, *args, **kwargs) == inspect.getcallargs(function,
↪ *args, **kwargs) == dict(a = 1, b = 1)
>>> args = (1,2); kwargs = {}
>>> assert getcallargs(function, *args, **kwargs) == inspect.getcallargs(function,
↪ *args, **kwargs) == dict(a = 1, b = 2)
>>> args = (1,); kwargs = {'b' : 2}
>>> assert getcallargs(function, *args, **kwargs) == inspect.getcallargs(function,
↪ *args, **kwargs) == dict(a = 1, b = 2)
```

pyg.base._inspect.**kwargs2args**(*function*, *args*, *kwargs*)

converts a list of paramters that were provided as kwargs, into args

Example

```
>>> assert kwargs2args(lambda a, b: a+b, (), dict(a = 1, b=2)) == ([1,2], {})
```

Parameters

function : callable args : tuple

parameters of function.

kwargs [dict] key-word parameters of function.

Returns

tuple a pair of a function args, kwargs.

CHAPTER

THREE

PYG.MONGO

A few words on MongoDB, a no-SQL database versus SQL:

- Mongo has 'collections' that are the equivalent of tables
- Mongo will refer to 'documents' instead of traditional records. Those records are unstructured and look like trees: dicts of dicts. They contain arbitary objects as well as just the primary types a SQL database is designed to support.
- Mongo collections do not have the concept of primary keys
- Mongo WHERE SQL clause is replaced by a query in a form of a dict "presented" to the collection object.
- Mongo SELECT SQL clause is replaced by a 'projection' on the cursor, specifying what fields are retrieved.

3.1 Query generator

We start by simplifying the way we generate mongo query dictionaries.

3.1.1 q and Q

class pyg.mongo._q.**Q**(*keys=None*)

The MongoDB interface for query of a collection (table) is via a creation of a complicated looking dict: https://docs.mongodb.com/manual/tutorial/query-documents/

This is rather complicated for the average user so Q simplifies it greatly. Q is based on TinyDB and users of TinyDB will recognise it. https://tinydb.readthedocs.io/en/latest/usage.html

q is the singleton of Q.

q supports both *calling* to generate the querying dict

```
>>> q(a = 1, b = 2)
```

or

```
>>> (q.a == 1) & (q.b == 2)   # {"$and": [{"a": {"$eq": 1}}, {"b": {"$eq": 2}}]}
>>> (q.a == 1) | (q.b == 2)   # {"$or": [{"a": {"$eq": 1}}, {"b": {"$eq": 2}}]}
```

or indeed

```
>>> q(q.a == 1, q.b  == 2)
```

Example

```
>>> from pyg import q
>>> import re
```

```
>>> assert dict(q.a == 1) == {"a": {"$eq": 1}}
>>> assert dict(q(a = [1,2])) == {'a': {'$in': [1, 2]}}
>>> assert dict(q(q.a == [1,2], q.b > 3)) == {'$and': [{"a": {"$in": [1, 2]}}, {"b
↪": {"$gt": 3}}]}  # a in [1,2] and b greater than 3
>>> assert dict(q(a = re.compile('^hello'))) == {'a': {'regex': '^hello'}}     #␣
↪a regex query using regex expressions
```

```
>>> assert dict(q.a.exists + q.b.not_exists)  == {"$and": [{"a": {"$exists": true}
↪}, {"b": {"$exists": false}}]}
```

```
>>> assert dict(~(q.a==1))  == {'$not': {"a": {"$eq": 1}}}
```

3.2 Tables in Mongo

3.2.1 mongo_cursor

mongo_cursor has hybrid functionality of a Mongo cursor and Mongo collection objects.

class pyg.mongo._cursor.**mongo_cursor**(*cursor*, *writer=None*, *reader=None*, *query=None*, ***_*)

mongo_cursor is a souped-up combination of mongo.Cursor and mongo.Collection with a simple API.

Parameters

cursor : MongoDB cursor or MongoDB collection

writer [True/False/string, optional] The default is None.

writer determines how data is written onto Mongo. MongoDB is great for manipulating/searching dict keys/values. If set to None, dataframes will be converted seamlessly to bytes and stored in MongoDB. Most often, this is fine.

At times, the actual dataframes in each doc, we may want to save in a file system. This may be because: - The DataFrames are stored as bytes in MongoDB anyway, so they are not searchable - Storing in files allows other non-python/non-MongoDB users easier access, allowing data to be detached from app - MongoDB free version has limitations on size of document - file based system may be faster - for data licensing issues, data must not sit on servers but stored on local computer - appending is tricky for bytes in Mongo but is relatively easy to do in both .npy and .parquet files:

- If you use AWS, you can use awswrangler to append messages to a timeseries

https://stackoverflow.com/questions/47191675/pandas-write-dataframe-to-parquet-format-with-append - For numpy appending, https://github.com/xor2k/npy-append-array/ will handle appending of messages for you

Therefore, if you set writer to .csv or .parquet, dataframes within will be saved to files first and we store in mongo references to these files.

For this to work, you need to tell us WHERE to store each document and this is how it works: If your document are primary-keyed by name, surname. Then… you can set the root centrally using expression like writer = 'c:/%name%surname.parquet'

Alternatively, writer = '.parquet' will encode only documents for which a 'root' key exists. This defers the decision of how to store itself to the cell.

reader [callable or None, optional] The default is None, using decode. Use reader = False to passthru

query [dict, optional] This is used to specify the Mongo query, e.g. q.a==1.

**_ :

Example

```
>>> from pyg import *
>>> cursor = mongo_table('test', 'test')
>>> cursor.drop()
```

insert some data

```
>>> table = dictable(a = range(5)) * dictable(b = range(5))
>>> cursor.insert_many(table)
>>> cursor.set(c = lambda a, b: a * b)
```

Filtering

```
>>> assert len(cursor) == 25
>>> assert len(cursor.find(a = 3)) == 5
>>> assert len(cursor.exc(a = 3)) == 20
>>> assert len(cursor.find(a = [3,2]).find(q.b<3)) == 6 ## can chain queries as␣
↪well as use q to create complicated expressions
```

Roww access

```
>>> cursor[0]
```

{'_id': ObjectId('603aec85cd15e2c090c07b87'), 'a': 0, 'b': 0}

```
>>> cursor[::] - '_id' == dictable(cursor) - '_id'
```

```
>>> dictable[25 x 3]
>>> a|b|c
>>> 0|0|0
>>> 0|1|0
>>> 0|2|0
>>> ...25 rows...
>>> 4|2|8
>>> 4|3|12
>>> 4|4|16
```

Column access

```
>>> cursor[['a', 'b']]  ## just columns 'a' and 'b'
>>> del cursor['c'] ## delete all c
>>> cursor.set(c = lambda a, b: a * b)
>>> assert cursor.find_one(a = 3, b = 2)[0].c == 6
```

Since mongo_cursor is too powerful, we also have a mongo_reader version which is read-only.

delete_many()

Equivalent to drop: deletes all documents the cursor currently points to.

Note

If you want to drop a subset of the data, then use c.find(criteria).delete_many()

Returns

itself

delete_one (**args*, ***kwargs*)
drops a specific record after verifying exactly one exists.

Parameters

*args : query **kwargs : query

Returns

itself

drop ()
Equivalent to drop: deletes all documents the cursor currently points to.

Note

If you want to drop a subset of the data, then use c.find(criteria).delete_many()

Returns

itself

insert_many (*table*)
inserts multiple documents into the collection

table [sequence of documents] list of dicts or dictable

mongo_cursor

Example simple insertion

```
>>> from pyg import *
>>> t = mongo_table('test', 'test')
>>> t = t.drop()
>>> values = dictable(a = [1,2,3,4,], b = [5,6,7,8])
>>> t = t.insert_many(values)
>>> t[::]
```

```
>>> dictable[4 x 3]
>>> _id                     |a|b
>>> 602daee68c336f6429a77bdd|1|5
>>> 602daee68c336f6429a77bde|2|6
>>> 602daee68c336f6429a77bdf|3|7
>>> 602daee68c336f6429a77be0|4|8
```

Example update

```
>>> table = t[::]
>>> modified = table(b = lambda b: b**2)
>>> t = t.insert_many(modified)
```

Since each of the documents we uploaded already has an _id…

```
>>> assert len(t) == 4
>>> t[::]
>>> dictable[4 x 3]
```

(continues on next page)

(continued from previous page)

```
>>> _id                     |a|b
>>> 602daee68c336f6429a77bdd|1|25
>>> 602daee68c336f6429a77bde|2|36
>>> 602daee68c336f6429a77bdf|3|49
>>> 602daee68c336f6429a77be0|4|64
```

insert_one (*doc*)

inserts/updates a single document.

If the document ALREADY has _id in it, it updates that document If the document has no _id in it, it inserts it as a new document

Parameters

doc [dict] document.

Example

```
>>> from pyg import *
>>> t = mongo_table('test', 'test')
>>> t = t.drop()
>>> values = dictable(a = [1,2,3,4,], b = [5,6,7,8])
>>> t = t.insert_many(values)
```

Example used to update an existing document

```
>>> doc = t[0]
>>> doc['c'] = 8
>>> str(doc)
>>> "{'_id': ObjectId('602d36150a5cd32717323197'), 'a': 1, 'b': 5, 'c': 8}"
```

```
>>> t = t.insert_one(doc)
>>> assert len(t) == 4
>>> assert t[0] == doc
```

Example used to insert

```
>>> doc = Dict(a = 1, b = 8, c = 10)
>>> t = t.insert_one(doc)
>>> assert len(t) == 5
>>> t.drop()
```

property raw

returns an unfiltered mongo_reader

set (***kwargs*)

updates all documents in current cursor based on the kwargs. It is similar to update_many but supports also functions

Parameters

kwargs: dict of values to be updated

Example

```
>>> from pyg import *
>>> t = mongo_table('test', 'test')
>>> t = t.drop()
>>> values = dictable(a = [1,2,3,4,], b = [5,6,7,8])
>>> t = t.insert_many(values)
>>> assert t[::]-'_id' == values
```

```
>>> t.set(c = lambda a, b: a+b)
>>> assert t[::]-'_id' == values(c = [6,8,10,12])
>>> t.set(d = 1)
>>> assert t[::]-'_id' == values(c =lambda a, b: a+b)(d = 1)
```

Returns

itself

update_many (*doc*, *upsert=False*)

updates all documents in current cursor based on the doc. The two are equivalent:

```
>>> cursot.update_many(doc)
>>> collection.update_many(cursor.spec, { 'set' : update})
```

Parameters

doc : dict of values to be updated

Returns

itself

update_one (*doc*, *upsert=True*)

- updates a document if an _id is present in doc.
- insert a document if an _id is not present and upsert is true

Parameters

doc [document] doc to be upserted.

upsert [bool, optional] insert if no document present? The default is True.

Returns

doc document updated.

3.2.2 mongo_reader

mongo_reader is a read-only version of the cursor to avoid any unintentional damage to database.

class pyg.mongo._reader.**mongo_reader** (*cursor*, *writer=None*, *reader=None*, *query=None*, ***_*)

mongo_reader is a read-only version of the mongo_cursor. You can instantiate it with a mongo_reader(cursor) call where cursor can be a mongo_cursor, a pymongo.Cursor or a pymongo.Collection

property address

Returns

tuple A unique combination of the client addres, database name and collection name, identifying the collection uniquely.

clone (***params*)

Returns

mongo_reader Returns a cloned version of current mongo_reader but allows additional parameters to be set (see spec and project)

property collection

Returns

pymongo.Collection object

count ()

cursor.count() and len(cursor) are the same and return the number of documents matching current specification.

distinct (*key*)

returns the distinct values of the key

key [str] a key in the documents.

list of strings distinct values

Example

```
>>> from pyg import *; import pymongo
>>> table = pymongo.MongoClient()['test']['test']
>>> table.insert_one(dict(name = 'alan', surname = 'abrahams', age = 39,␣
↪marriage = dt(2000)))
>>> table.insert_one(dict(name = 'barbara', surname = 'brown', age = 50,␣
↪marriage = dt(2020)))
>>> table.insert_one(dict(name = 'charlie', surname = 'cohen', age = 20))
```

```
>>> t = mongo_reader(table)
>>> assert t.name == t.distinct('name') == ['alan', 'barbara', 'charlie']
>>> table.drop()
```

docs (*doc='doc'*, **keys*)

self[::] flattens the entire document. At times, we want to see the full documents, indexed by keys and docs does that. returns a dictable with both keys and the document in the 'doc' column

exc (***kwargs*)

filters 'negatively' removing documents that match the criteria specified.

cursor filtered documents.

Example

```
>>> from pyg import *; import pymongo
>>> table = pymongo.MongoClient()['test']['test']
>>> table.insert_one(dict(name = 'alan', surname = 'abrahams', age = 39,␣
↪marriage = dt(2000)))
```

(continues on next page)

(continued from previous page)

```
>>> table.insert_one(dict(name = 'barbara', surname = 'brown', age = 50,␣
↪marriage = dt(2020)))
>>> table.insert_one(dict(name = 'charlie', surname = 'cohen', age = 20))
```

```
>>> t = mongo_reader(table)
>>> assert len(t.exc(name = 'alan')) == 2
>>> assert len(t.exc(name = ['alan', 'barbara'])) == 1
>>> table.drop()
```

find(**args*, ***kwargs*)

Same as self.specify()

The 'spec' is the cursor's filter on documents (can think of it as row-selection) within the collection. We use q (see pyg.mongo._q.q) to specify the filter on the cursor.

Returns

A filtered mongo_reader cursor

Example

```
>>> from pyg import *; import pymongo
>>> table = pymongo.MongoClient()['test']['test']
>>> table.insert_one(dict(name = 'alan', surname = 'abrahams', age = 39,␣
↪marriage = dt(2000)))
>>> table.insert_one(dict(name = 'barbara', surname = 'brown', age = 50,␣
↪marriage = dt(2020)))
>>> table.insert_one(dict(name = 'charlie', surname = 'cohen', age = 20))
```

```
>>> t = mongo_reader(table)
>>> assert len(t.find(name = 'alan')) == 1
>>> assert len(t.find(q.age>25)) == 2
>>> assert len(t.find(q.age>25, q.marriage<dt(2010))) == 1
```

```
>>> table.drop()
```

find_one(*doc=None*, **args*, ***kwargs*)

searches for records based either on the doc, or the args/kwargs specified. Unlike mongo cursor which finds one of many, here, when we ask for find_one, we will throw an exception if more than one documents are found.

Returns

A cursor pointing to a single record (document)

inc(**args*, ***kwargs*)

Same as self.specify()

The 'spec' is the cursor's filter on documents (can think of it as row-selection) within the collection. We use q (see pyg.mongo._q.q) to specify the filter on the cursor.

Returns

A filtered mongo_reader cursor

Example

```
>>> from pyg import *; import pymongo
>>> table = pymongo.MongoClient()['test']['test']
>>> table.insert_one(dict(name = 'alan', surname = 'abrahams', age = 39,␣
↪marriage = dt(2000)))
>>> table.insert_one(dict(name = 'barbara', surname = 'brown', age = 50,␣
↪marriage = dt(2020)))
>>> table.insert_one(dict(name = 'charlie', surname = 'cohen', age = 20))
```

```
>>> t = mongo_reader(table)
>>> assert len(t.find(name = 'alan')) == 1
>>> assert len(t.find(q.age>25)) == 2
>>> assert len(t.find(q.age>25, q.marriage<dt(2010))) == 1
```

```
>>> table.drop()
```

project(*projection=None*)

The 'projection' is the cursor's column selection on documents. If in SQL we write SELECT col1, col2 FROM . . . , in Mongo, the cursor.projection = ['col1', 'col2']

Parameters

projection: a list/str of keys we are interested in reading. Note that nested keys are OK: 'level1.level2.name' is perfectly good

Returns

A mongo_reader cursor filtered to read just these keys

property projection

Returns

The 'projection' is the cursor's column selection on documents. If in SQL we write SELECT col1, col2 FROM . . . , in Mongo, the cursor.projection = ['col1', 'col2']

property raw

returns an unfiltered mongo_reader

read(*item=0, reader=None*)

reads the next document from the collection.

Parameters

item [int, optional] Please read the ith record. The default is 0.

reader [callable/list of callables, optional] When we read the document from the collection, we first transform them. The default behaviour is to use pyg.base._encode.decode but you may pass reader = False to grab the raw data from mongo

Returns

document The document from Mongo

sort(**by*)

sorting on server side, per key(s)

by : str/list of strs

sorted cursor.

property spec

Returns

The 'spec' is the cursor's filter on documents (can think of it as row-selection) within the collection

specify(**args*, ***kwargs*)

The 'spec' is the cursor's filter on documents (can think of it as row-selection) within the collection. We use q (see pyg.mongo._q.q) to specify the filter on the cursor.

Returns

A filtered mongo_reader cursor

3.2.3 mongo_pk_reader

mongo_pk_reader extends the standard reader to handle tables with primary keys (pk) while being read-only.

class pyg.mongo._pk_reader.**mongo_pk_reader**(*cursor*, *pk*, *writer=None*, *reader=None*, *query=None*, ***_*)

we set up a system in Mongo to ensure we can mimin tables with primary keys. The way we do this is two folds:

- At document insertion, we mark as _deleted old documents sharing the same keys by adding a key _deleted to the old doc
- At reading, we filter for documents where q._deleted.not_exists.

clone(***kwargs*)

Returns

mongo_reader Returns a cloned version of current mongo_reader but allows additional parameters to be set (see spec and project)

create_index(**keys*)

creates a sorted index on the collection

Parameters

***keys** [strings] if misssing, use the primary keys.

dedup()

Although in principle, if a single process reads/writes to Mongo, we should not get duplicates. In practice, when multiple clients access the database, we occasionally get multiple records with the same primary keys. When this happens, we also end up with poor mongo _ids

mongo_pk_cursor Hopefully, a table with unique keys.

docs(*doc='doc'*, **keys*)

self[::] flattens the entire document. At times, we want to see the full documents, indexed by keys and docs does that. returns a dictable with both keys and the document in the 'doc' column

3.2.4 mongo_pk_cursor

mongo_pk_cursor is our go-to object and it manages all our primary-keyed tables. .. autoclass:: pyg.mongo._pk_cursor.mongo_pk_cursor

members

3.3 encoding docs before saving to mongo

Before we save data to Mongo, we may need to transform it, especially if we are to save pd.DataFrame. By default, we encode them into bytes and push to mongo. You can choose to save pandas dataframes/series as .parquet files and numpy arrays into .npy files.

3.3.1 parquet_write

`pyg.mongo._encoders.`**`parquet_write`**(*doc*, *root=None*)

MongoDB is great for manipulating/searching dict keys/values. However, the actual dataframes in each doc, we may want to save in a file system. - The DataFrames are stored as bytes in MongoDB anyway, so they are not searchable - Storing in files allows other non-python/non-MongoDB users easier access, allowing data to be detached from app - MongoDB free version has limitations on size of document - file based system may be faster, especially if saved locally not over network - for data licensing issues, data must not sit on servers but stored on local computer

Therefore, the doc encode will cycle through the elements in the doc. Each time it sees a pd.DataFrame/pd.Series, it will - determine where to write it (with the help of the doc) - save it to a .parquet file

3.3.2 csv_write

`pyg.mongo._encoders.`**`csv_write`**(*doc*, *root=None*)

MongoDB is great for manipulating/searching dict keys/values. However, the actual dataframes in each doc, we may want to save in a file system. - The DataFrames are stored as bytes in MongoDB anyway, so they are not searchable - Storing in files allows other non-python/non-MongoDB users easier access, allowing data to be detached from orignal application - MongoDB free version has limitations on size of document - file based system may be faster, especially if saved locally not over network - for data licensing issues, data must not sit on servers but stored on local computer

Therefore, the doc encode will cycle through the elements in the doc. Each time it sees a pd.DataFrame/pd.Series, it will - determine where to write it (with the help of the doc) - save it to a .csv file

3.4 cells in Mongo

Now that we have a database, we construct cells that can load/save data to collections.

3.4.1 db_cell

class pyg.mongo._db_cell.**db_cell**(*function=None*, *output=None*, *db=None*, ***kwargs*)

a db_cell is a specialized cell with a 'db' member pointing to a database where cell is to be stored. We use this to implement save/load for the cell.

It is important to recognize the duality in the design: - the job of the cell.db is to be able to save/load based on the primary keys. - the job of the cell is to provide the primary keys to the db object.

The cell saves itself by 'presenting' itself to cell.db() and say… go on, load my data based on my keys.

Example saving & loading

```
>>> from pyg import *
>>> people = partial(mongo_table, db = 'test', table = 'test', pk = ['name',
↪'surname'])
>>> anna = db_cell(db = people, name = 'anna', surname = 'abramzon', age = 46).
↪save()
>>> bob  = db_cell(db = people, name = 'bob', surname = 'brown', age = 25).save()
>>> james = db_cell(db = people, name = 'james', surname = 'johnson', age = 39).
↪save()
```

Now we can pull the data directly from the database

```
>>> people()['name', 'surname', 'age'][::]
>>> dictable[3 x 4]
>>> _id                     |age|name |surname
>>> 601e732e0ef13bec9cd8a6cb|39 |james|johnson
>>> 601e73db0ef13bec9cd8a6d4|46 |anna |abramzon
>>> 601e73db0ef13bec9cd8a6d7|25 |bob  |brown
```

db_cell can implement a function:

```
>>> def is_young(age):
>>>     return age < 30
>>> bob.function = is_young
>>> bob = bob.go()
>>> assert bob.data is True
```

When run, it saves its new data to Mongo and we can load its own data:

```
>>> new_cell_with_just_db_and_keys = db_cell(db = people, name = 'bob', surname =
↪'brown')
>>> assert 'age' not in new_cell_with_just_db_and_keys
>>> now_with_the_data_from_database = new_cell_with_just_db_and_keys.load()
>>> assert now_with_the_data_from_database.age == 25
```

```
>>> people()['name', 'surname', 'age', 'data'][::]
>>>  dictable[3 x 4]
>>> _id                     |age|name |surname |data
>>> 601e732e0ef13bec9cd8a6cb|39 |james|johnson |None
>>> 601e73db0ef13bec9cd8a6d4|46 |anna |abramzon|None
>>> 601e73db0ef13bec9cd8a6d7|25 |bob  |brown   |True
>>> people().raw.drop()
```

load(*mode=0*, *keys=None*)

loads a document from the database and updates various keys

mode [int , optional] if -1, then does not load and skips this function if 0, then will load if found. If not found, will return original document if 1, then will throw an exception if no document is found in the database The default is 0.

keys [str/list of str/True, optional] determines which additional keys (other than output) are loaded onto the existing cell from the saved one. output keys are always loaded.

document

3.4.2 periodic_cell

class pyg.mongo._periodic_cell.**periodic_cell**(*function=None*, *output=None*, *db=None*, *_period='1b'*, *updated=None*, ***kwargs*)

periodic_cell inherits from db_cell its ability to save itself in MongoDb using its db members Its calculation schedule depends on when it was last updated.

Example

```
>>> from pyg import *
>>> c = periodic_cell(lambda a: a + 1, a = 0)
```

We now assert it needs to be calculated and calculate it...

```
>>> assert c.run()
>>> c = c.go()
>>> assert c.data == 1
>>> assert not c.run()
```

Now let us cheat and tell it, it was last run 3 days ago...

```
>>> c.updated = dt(-3)
>>> assert c.run()
```

3.4.3 get_cell

3.4.4 db_save

pyg.mongo._db_cell.**db_save**(*value*)

saves a db_cell from the database. Will iterates through lists and dicts

Parameters

value: obj db_cell (or list/dict of) to be loaded

Example

```
>>> from pyg import *
>>> db = partial(mongo_table, table = 'test', db = 'test', pk = ['a','b'])
>>> c = db_cell(add_, a = 2, b = 3, key = 'test', db = db)
>>> c = db_save(c)
>>> assert get_cell('test', 'test', a = 2, b = 3).key == 'test'
```

3.4.5 db_load

`pyg.mongo._db_cell.`**`db_load`**`(`*value*, *mode=0*`)`

loads a db_cell from the database. Iterates through lists and dicts

Parameters

value: obj db_cell (or list/dict of) to be loaded

mode: int loading mode -1: dont load, +1: load and throw an exception if not found, 0: load if found

3.4.6 db_ref

`pyg.mongo._db_cell.`**`db_ref`**`(`*value*`)`

db_ref strips a db_cell so that it contains only the reference needed to its location in the database. When we save OTHER cells, referencing this cell, we apply db_ref and only save the bare data needed to reload cell

Example

```
>>> from pyg import *
>>> db = partial(mongo_table, table = 'test', db = 'test', pk = 'key')
>>> c = db_cell(add_, a = 1, b = 2, key = 'key', db = db)()
>>> assert c.data == 3
```

```
>>> bare = db_ref(c)
>>> assert 'a' not in bare and 'b' not in bare and 'data' not in bare
```

```
>>> reloaded = db_load(bare)
>>> assert reloaded.a == 1 and reloaded.data == 3
```

Parameters

value: obj db_cell (or list/dict of) to be made into reference

CHAPTER

FOUR

PYG.TIMESERIES

Given pandas, why do we need this timeseries library? pandas is amazing but there are a few features in pyg.timeseries designed to enhance it. There are three issues with pandas that pyg.timeseries tries to address:

- pandas works on pandas objects (obviously) but not on numpy arrays.
- **pandas handles TimeSeries with nan inconsistently across its functions. This makes your results sensitive to reindexing/res**
  - a.expanding() & a.ewm() **ignore** nan's for calculation and then ffill the result.
  - a.diff(), a.rolling() **include** any nans in the calculation, leading to nan propagation.
- pandas is great if you have the full timeseries. However, if you now want to run the same calculations in a live environment, on recent data, pandas cannot help you: you have to stick the new data at the end of the DataFrame and rerun.

pyg.timeseries tries to address this:

- pyg.timeseries agrees with pandas 100% on DataFrames (with no nan) while being of comparable (if not faster) speed
- pyg.timeseries works seemlessly on pandas objects and on numpy arrays, with no code change.
- pyg.timeseries handles nan consistently across all its functions, 'ignoring' all nan, making your results consistent regardless of resampling.
- **pyg.timeseries exposes the state of the internal function calculation. The exposure of internal states allows us to calculate t**
  - risk calculations, Monte Carlo scenarios: We can run a trading strategy up to today and then generate multiple scenarios and see what-if, without having to rerun the full history.
  - live versus history: pandas is designed to run a full historical simulation. However, once we reach "today", speed is of the essense and running a full historical simulation every time we ingest a new price, is just too slow. That is why most fast trading is built around fast state-machines. Of course, making sure research & live versions do the same thing is tricky. pyg gives you the ability to run two systems in parallel with almost the same code base: run full history overnight and then run today's code base instantly, instantiated with the output of the historical simulation.

4.1 simple functions

4.1.1 diff

pyg.timeseries._rolling.**diff**(*a*, *n=1*, *axis=0*, *data=None*, *state=None*)
equivalent to a.diff(n) in pandas if there are no nans. If there are, we SKIP nans rather than propagate them.

Example : matching pandas no nan's

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> assert eq(timer(diff, 1000)(a), timer(lambda a, n=1: a.diff(n), 1000)(a))
```

Example : nan skipping

```
>>> a = np.array([1., np.nan, 3., 9.])
>>> assert eq(diff(a),                         np.array([np.nan, np.nan, 2.0,   6.
↪0]))
>>> assert eq(pd.Series(a).diff().values,   np.array([np.nan, np.nan, np.nan,6.
↪0]))
```

4.1.2 shift

pyg.timeseries._rolling.**shift**(*a*, *n=1*, *axis=0*, *data=None*, *state=None*)
Equivalent to a.shift() with support to arra

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

n: int size of rolling window

Example

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series([1.,2,3,4,5], drange(-4))
>>> assert eq(shift(a), pd.Series([np.nan,1,2,3,4], drange(-4)))
>>> assert eq(shift(a,2), pd.Series([np.nan,np.nan,1,2,3], drange(-4)))
>>> assert eq(shift(a,-1), pd.Series([2,3,4,5,np.nan], drange(-4)))
```

Example np.ndarrays

```
>>> assert eq(shift(a.values), shift(a).values)
```

Example nan skipping

```
>>> a = pd.Series([1.,2,np.nan,3,4], drange(-4))
>>> assert eq(shift(a), pd.Series([np.nan,1,np.nan, 2,3], drange(-4)))
>>> assert eq(a.shift(), pd.Series([np.nan,1,2,np.nan,3], drange(-4))) # the
↪location of the nan changes
```

Example state management

```
>>> old = a.iloc[:3]
>>> new = a.iloc[3:]
>>> old_ts = shift_(old)
>>> new_ts = shift(new, **old_ts)
>>> assert eq(new_ts, shift(a).iloc[3:])
```

4.1.3 ratio

`pyg.timeseries._rolling.ratio(a, n=1, data=None, state=None)`

Equivalent to a.diff() but in log-space..

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

n: int size of rolling window

Example

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series([1.,2,3,4,5], drange(-4))
>>> assert eq(ratio(a), pd.Series([np.nan, 2, 1.5, 4/3,1.25], drange(-4)))
>>> assert eq(ratio(a,2), pd.Series([np.nan, np.nan, 3, 2, 5/3], drange(-4)))
```

4.1.4 ts_count

`pyg.timeseries._ts.ts_count(a)` *is equivalent to a.count()*

- supports numpy arrays
- handles nan
- supports state management
- pandas is actually faster on count

```
>>> # create sample data:
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999)); a[a>0] = np.nan
```

Example pandas matching

```
>>> assert ts_count(a) == a.count()
```

Example numpy

```
>>> assert ts_count(a.values) == ts_count(a)
```

Example state management

```
>>> old = ts_count_(a.iloc[:2000])
>>> new = ts_count(a.iloc[2000:], vec = old.vec)
>>> assert new == ts_count(a)
```

4.1.5 ts_sum

pyg.timeseries._ts.**ts_sum**(*a*) *is equivalent to a.sum()*

- supports numpy arrays
- handles nan
- supports state management
- pandas is actually faster on count

```
>>> # create sample data:
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999)); a[a>0] = np.nan
```

Example pandas matching

```
>>> assert ts_sum(a) == a.sum()
```

Example numpy

```
>>> assert ts_sum(a.values) == ts_sum(a)
```

Example state management

```
>>> old = ts_sum_(a.iloc[:2000])
>>> new = ts_sum(a.iloc[2000:], vec = old.vec)
>>> assert new == ts_sum(a)
```

4.1.6 ts_mean

pyg.timeseries._ts.**ts_mean**(*a*) *is equivalent to a.mean()*

- supports numpy arrays
- handles nan
- supports state management
- pandas is actually faster on count

```
>>> # create sample data:
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999)); a[a>0] = np.nan
```

Example pandas matching

```
>>> assert ts_mean(a) == a.mean()
```

Example numpy

```
>>> assert ts_mean(a.values) == ts_mean(a)
```

Example state management

```
>>> old = ts_mean_(a.iloc[:2000])
>>> new = ts_mean(a.iloc[2000:], vec = old.vec)
>>> assert new == ts_mean(a)
```

4.1.7 ts_rms

pyg.timeseries._ts.**ts_rms**(*a*, *axis=0*, *data=None*, *state=None*)

ts_rms(a) is equivalent to (a**2).mean()**0.5

- supports numpy arrays
- handles nan
- supports state management

```
>>> # create sample data:
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999)); a[a>0] = np.nan
```

Example pandas matching

```
>>> assert abs(ts_std(a) - a.std())<1e-13
```

Example numpy

```
>>> assert ts_std(a.values) == ts_std(a)
```

Example state management

```
>>> old = ts_rms_(a.iloc[:2000])
>>> new = ts_rms(a.iloc[2000:], vec = old.vec)
>>> assert new == ts_rms(a)
```

4.1.8 ts_std

pyg.timeseries._ts.**ts_std**(*a*) *is equivalent to a.std()*

- supports numpy arrays
- handles nan
- supports state management

```
>>> # create sample data:
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999)); a[a>0] = np.nan
```

Example pandas matching

```
>>> assert abs(ts_std(a) - a.std())<1e-13
```

Example numpy

```
>>> assert ts_std(a.values) == ts_std(a)
```

Example state management

```
>>> old = ts_std_(a.iloc[:2000])
>>> new = ts_std(a.iloc[2000:], vec = old.vec)
>>> assert new == ts_std(a)
```

4.1.9 ts_skew

pyg.timeseries._ts.**ts_skew**(*a*, *0*) *is equivalent to a.skew()*

- supports numpy arrays
- handles nan
- faster than pandas
- supports state management

```
>>> # create sample data:
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999)); a[a>0] = np.nan
```

Example pandas matching

```
>>> assert abs(ts_skew(a, 0) - a.skew())<1e-13
```

Example numpy

```
>>> assert ts_skew(a.values) == ts_skew(a)
```

Example state management

```
>>> old = ts_skew_(a.iloc[:2000])
>>> new = ts_skew(a.iloc[2000:], vec = old.vec)
>>> assert new == ts_skew(a)
```

4.1.10 ts_min

pyg.timeseries._ts.**ts_max**(*a) is equivalent to pandas a.min()*

4.1.11 ts_max

pyg.timeseries._ts.**ts_max**(*a) is equivalent to pandas a.min()*

4.1.12 ts_median

pyg.timeseries._ts.**ts_median**(*a*, *axis=0*)

4.1.13 fnna

pyg.timeseries._rolling.**fnna**(*a*, *n=1*, *axis=0*)

returns the index in a of the nth first non-nan.

Parameters

a : array/timeseries n: int, optional, default = 1

Example

```
>>> a = np.array([np.nan,np.nan,1,np.nan,np.nan,2,np.nan,np.nan,np.nan])
>>> fnna(a,n=-2)
```

4.1.14 v2na/na2v

pyg.timeseries._rolling.**v2na**(*a*, *old=0.0*, *new=nan*)

replaces an old value with a new value (default is nan)

Examples

```
>>> from pyg import *
>>> a = np.array([1., np.nan, 1., 0.])
>>> assert eq(v2na(a), np.array([1., np.nan, 1., np.nan]))
>>> assert eq(v2na(a,1), np.array([np.nan, np.nan, np.nan, 0]))
>>> assert eq(v2na(a,1,0), np.array([0., np.nan, 0., 0.]))
```

Parameters

a : array/timeseries old: float

value to be replaced

new [float, optional] new value to be used, The default is np.nan.

Returns

array/timeseries

pyg.timeseries._rolling.**na2v**(*a*, *new=0.0*)
replaces a nan with a new value

Example

```
>>> from pyg import *
>>> a = np.array([1., np.nan, 1.])
>>> assert eq(na2v(a), np.array([1., 0.0, 1.]))
>>> assert eq(na2v(a,1), np.array([1., 1., 1.]))
```

Parameters

a : array/timeseries new : float, optional

DESCRIPTION. The default is 0.0.

Returns

array/timeseries

4.1.15 ffill/bfill

pyg.timeseries._rolling.**ffill**(*a*, *n=0*, *axis=0*, *data=None*, *state=None*)
returns a forward filled array, up to n values forward. supports state manegement which is needed if we want only nth

Example

```
>>> a = np.array([np.nan,np.nan,1,np.nan,np.nan,2,np.nan,np.nan,np.nan])
>>> fnna(a, n=-2)
```

pyg.timeseries._rolling.**bfill**(*a*, *n=- 1*, *axis=0*)
equivalent to a.fillna('bfill'). There is no state-aware as this function is forward looking

Example

```
>>> from pyg import *
>>> a = np.array([np.nan, 1., np.nan])
>>> b = np.array([1., 1., np.nan])
>>> assert eq(bfill(a), b)
```

Example pd.Series

```
>>> ts = pd.Series(a, drange(-2))
>>> assert eq(bfill(ts).values, b)
```

4.1.16 nona

`pyg.timeseries._ts.nona(df, value=nan)`

removes rows that are entirely nan (or a specific other value)

Parameters

df : dataframe/ndarray

value [float, optional] value to be removed. The default is np.nan.

Example

```
>>> from pyg import *
>>> a = np.array([1,np.nan,2,3])
>>> assert eq(nona(a), np.array([1,2,3]))
```

Example multiple columns

```
>>> a = np.array([[1,np.nan,2,np.nan], [np.nan, np.nan, np.nan, 3]]).T
>>> b = np.array([[1,2,np.nan], [np.nan, np.nan, 3]]).T ## 2nd row has nans across
>>> assert eq(nona(a), b)
```

4.2 expanding window functions

4.2.1 expanding_mean

`pyg.timeseries._expanding.expanding_mean(a, axis=0, data=None, state=None)`

equivalent to pandas a.expanding().mean().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

axis [int, optional] 0/1/-1. The default is 0.

t0,t1,data: state parameters to instantiate the calculation. t0 = total points so far, t2 = total(a) so far

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.expanding().mean(); ts = expanding_mean(a)
>>> assert eq(ts,panda)
```

Example nan handling

Unlike pandas, timeseries does not forward fill the nans.

```
>>> a[a<0.1] = np.nan
>>> panda = a.expanding().mean(); ts = expanding_mean(a)
```

```
>>> pd.concat([panda,ts], axis=1)
>>>                    0         1
>>> 1993-09-23  1.562960  1.562960
>>> 1993-09-24  0.908910  0.908910
>>> 1993-09-25  0.846817  0.846817
>>> 1993-09-26  0.821423  0.821423
>>> 1993-09-27  0.821423       NaN
>>>                  ...       ...
>>> 2021-02-03  0.870358  0.870358
>>> 2021-02-04  0.870358       NaN
>>> 2021-02-05  0.870358       NaN
>>> 2021-02-06  0.870358       NaN
>>> 2021-02-07  0.870353  0.870353
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = expanding_mean(a)
>>> old_ts = expanding_mean_(old)
>>> new_ts = expanding_mean(new, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(expanding_mean(dict(x = a, y = a**2)), dict(x = expanding_mean(a),␣
↪y = expanding_mean(a**2)))
>>> assert eq(expanding_mean([a,a**2]), [expanding_mean(a), expanding_mean(a**2)])
```

4.2.2 expanding_rms

pyg.timeseries._expanding.**expanding_rms**(*a*, *axis=0*, *data=None*, *state=None*)

equivalent to pandas (a**2).expanding().mean()**0.5). - works with np.arrays - handles nan without forward filling. - supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame, list/dict of these] timeseries

axis [int, optional] 0/1/-1. The default is 0.

t0,t2,data: state parameters to instantiate the calculation. t0 = total points so far, t2 = total(x^2) so far

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = (a**2).expanding().mean()**0.5; ts = expanding_rms(a)
>>> assert eq(ts,panda)
```

Example nan handling

Unlike pandas, timeseries does not forward fill the nans.

```
>>> a[a<0.1] = np.nan
>>> panda = (a**2).expanding().mean()**0.5; ts = expanding_rms(a)
```

```
>>> pd.concat([panda,ts], axis=1)
>>>                    0         1
>>> 1993-09-23  0.160462  0.160462
>>> 1993-09-24  0.160462       NaN
>>> 1993-09-25  0.160462       NaN
>>> 1993-09-26  0.160462       NaN
>>> 1993-09-27  0.160462       NaN
>>>                  ...       ...
>>> 2021-02-03  1.040346  1.040346
>>> 2021-02-04  1.040346       NaN
>>> 2021-02-05  1.040338  1.040338
>>> 2021-02-06  1.040337  1.040337
>>> 2021-02-07  1.040473  1.040473
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = expanding_rms(a)
>>> old_ts = expanding_rms_(old)
>>> new_ts = expanding_rms(new, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(expanding_rms(dict(x = a, y = a**2)), dict(x = expanding_rms(a), y␣
↪= expanding_rms(a**2)))
>>> assert eq(expanding_rms([a,a**2]), [expanding_rms(a), expanding_rms(a**2)])
```

4.2.3 expanding_std

`pyg.timeseries._expanding.`**`expanding_std`**(*a*, *axis=0*, *data=None*, *state=None*)

equivalent to pandas a.expanding().std().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

axis [int, optional] 0/1/-1. The default is 0.

t0,t1,t2,data: state parameters to instantiate the calculation. t0,t1,t2= total(points),total(a),total(a**2) so far

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.expanding().std(); ts = expanding_std(a)
>>> assert abs(ts-panda).max()<1e-10
```

Example nan handling

Unlike pandas, timeseries does not forward fill the nans.

```
>>> a[a<0.1] = np.nan
>>> panda = a.expanding().std(); ts = expanding_std(a)
```

```
>>> pd.concat([panda,ts], axis=1)
>>>                    0         1
>>> 1993-09-23       NaN       NaN
>>> 1993-09-24       NaN       NaN
>>> 1993-09-25       NaN       NaN
>>> 1993-09-26       NaN       NaN
>>> 1993-09-27       NaN       NaN
>>>                  ...       ...
>>> 2021-02-03  0.590448  0.590448
>>> 2021-02-04  0.590448       NaN
>>> 2021-02-05  0.590475  0.590475
>>> 2021-02-06  0.590475       NaN
>>> 2021-02-07  0.590411  0.590411
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = expanding_std(a)
>>> old_ts = expanding_std_(old)
>>> new_ts = expanding_std(new, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(expanding_std(dict(x = a, y = a**2)), dict(x = expanding_std(a), y␣
↪= expanding_std(a**2)))
>>> assert eq(expanding_std([a,a**2]), [expanding_std(a), expanding_std(a**2)])
```

4.2.4 expanding_sum

`pyg.timeseries._expanding.`**`expanding_sum`**(*a*, *axis=0*, *data=None*, *state=None*)

equivalent to pandas a.expanding().sum().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

axis [int, optional] 0/1/-1. The default is 0.

t0,t1,data: state parameters to instantiate the calculation. t0 = total points so far, t2 = total(a) so far

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.expanding().sum(); ts = expanding_sum(a)
>>> assert eq(ts,panda)
```

Example nan handling

Unlike pandas, timeseries does not forward fill the nans.

```
>>> a[a<0.1] = np.nan
>>> panda = a.expanding().sum(); ts = expanding_sum(a)
```

```
>>> pd.concat([panda,ts], axis=1)
>>>                        0          1
>>> 1993-09-23           NaN        NaN
>>> 1993-09-24           NaN        NaN
>>> 1993-09-25      0.645944   0.645944
>>> 1993-09-26      2.816321   2.816321
>>> 1993-09-27      2.816321        NaN
>>>                      ...        ...
>>> 2021-02-03   3976.911348  3976.911348
>>> 2021-02-04   3976.911348        NaN
>>> 2021-02-05   3976.911348        NaN
>>> 2021-02-06   3976.911348        NaN
>>> 2021-02-07   3976.911348        NaN
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = expanding_sum(a)
>>> old_ts = expanding_sum_(old)
>>> new_ts = expanding_sum(new, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(expanding_sum(dict(x = a, y = a**2)), dict(x = expanding_sum(a), y␣
↪= expanding_sum(a**2)))
>>> assert eq(expanding_sum([a,a**2]), [expanding_sum(a), expanding_sum(a**2)])
```

4.2.5 expanding_skew

`pyg.timeseries._expanding.`**`expanding_skew`**(*a*, *bias=False*, *axis=0*, *data=None*, *state=None*)
equivalent to pandas a.expanding().skew() which doesn't exist

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

axis [int, optional] 0/1/-1. The default is 0.

t0,t1,t2,t3,data: state parameters to instantiate the calculation. t0,t1,t2= total(points),total(a),total(a**2),total(a**3) so far

Example state management

One can split the calculation and run old and new data separately.

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
```

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = expanding_skew(a)
>>> old_ts = expanding_skew_(old)
>>> new_ts = expanding_skew(new, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(expanding_skew(dict(x = a, y = a**2)), dict(x = expanding_skew(a),␣
↪y = expanding_skew(a**2)))
>>> assert eq(expanding_skew([a,a**2]), [expanding_skew(a), expanding_skew(a**2)])
```

4.2.6 expanding_min

pyg.timeseries._min.**expanding_min**(*a*, *axis=0*, *data=None*, *state=None*)

equivalent to pandas a.expanding().min().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

axis [int, optional] 0/1/-1. The default is 0.

m,data: state parameters to instantiate the calculation. m = min so far

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.expanding().min(); ts = expanding_min(a)
>>> assert eq(ts,panda)
```

Example nan handling

Unlike pandas, timeseries does not forward fill the nans.

```
>>> a[a<0.1] = np.nan
>>> panda = a.expanding().min(); ts = expanding_min(a)
```

```
>>> pd.concat([panda,ts], axis=1)
>>>                    0         1
>>> 1993-09-24       NaN       NaN
>>> 1993-09-25       NaN       NaN
>>> 1993-09-26  0.775176  0.775176
>>> 1993-09-27  0.691942  0.691942
>>> 1993-09-28  0.691942       NaN
>>>                  ...       ...
>>> 2021-02-04  0.100099  0.100099
>>> 2021-02-05  0.100099       NaN
>>> 2021-02-06  0.100099       NaN
>>> 2021-02-07  0.100099  0.100099
>>> 2021-02-08  0.100099  0.100099
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = expanding_min(a)
>>> old_ts = expanding_min_(old)
>>> new_ts = expanding_min(new, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(expanding_min(dict(x = a, y = a**2)), dict(x = expanding_min(a), y␣
↪= expanding_min(a**2)))
>>> assert eq(expanding_min([a,a**2]), [expanding_min(a), expanding_min(a**2)])
```

4.2.7 expanding_max

pyg.timeseries._max.**expanding_max**(*a*, *axis=0*, *data=None*, *state=None*)
equivalent to pandas a.expanding().max().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

axis [int, optional] 0/1/-1. The default is 0.

m,data: state parameters to instantiate the calculation. m = max so far

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.expanding().max(); ts = expanding_max(a)
>>> assert eq(ts,panda)
```

Example nan handling

Unlike pandas, timeseries does not forward fill the nans.

```
>>> a[a<0.1] = np.nan
>>> panda = a.expanding().max(); ts = expanding_max(a)
```

```
>>> pd.concat([panda,ts], axis=1)
>>>                    0         1
>>> 1993-09-24       NaN       NaN
>>> 1993-09-25       NaN       NaN
>>> 1993-09-26  0.875409  0.875409
>>> 1993-09-27  0.875409       NaN
>>> 1993-09-28  0.875409       NaN
>>>                  ...       ...
>>> 2021-02-04  3.625858  3.625858
>>> 2021-02-05  3.625858       NaN
>>> 2021-02-06  3.625858  3.625858
>>> 2021-02-07  3.625858       NaN
>>> 2021-02-08  3.625858       NaN
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = expanding_max(a)
>>> old_ts = expanding_max_(old)
>>> new_ts = expanding_max(new, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(expanding_max(dict(x = a, y = a**2)), dict(x = expanding_max(a), y␣
↪= expanding_max(a**2)))
>>> assert eq(expanding_max([a,a**2]), [expanding_max(a), expanding_max(a**2)])
```

4.2.8 expanding_median

pyg.timeseries._median.**expanding_median**(*a*, *axis=0*)

equivalent to pandas a.expanding().median().

- works with np.arrays
- handles nan without forward filling.
- There is no state-aware version since this requires essentially the whole history to be stored.

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

axis [int, optional] 0/1/-1. The default is 0.

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.expanding().median(); ts = expanding_median(a)
>>> assert eq(ts,panda)
```

Example nan handling

Unlike pandas, timeseries does not forward fill the nans.

```
>>> a[a<0.1] = np.nan
>>> panda = a.expanding().median(); ts = expanding_median(a)
```

```
>>> pd.concat([panda,ts], axis=1)
>>>                    0         1
>>> 1993-09-23  1.562960  1.562960
>>> 1993-09-24  0.908910  0.908910
>>> 1993-09-25  0.846817  0.846817
>>> 1993-09-26  0.821423  0.821423
>>> 1993-09-27  0.821423       NaN
>>>                  ...       ...
```

(continues on next page)

(continued from previous page)

```
>>> 2021-02-03  0.870358  0.870358
>>> 2021-02-04  0.870358       NaN
>>> 2021-02-05  0.870358       NaN
>>> 2021-02-06  0.870358       NaN
>>> 2021-02-07  0.870353  0.870353
```

Example dict/list inputs

```
>>> assert eq(expanding_median(dict(x = a, y = a**2)), dict(x = expanding_
↪median(a), y = expanding_median(a**2)))
>>> assert eq(expanding_median([a,a**2]), [expanding_median(a), expanding_
↪median(a**2)])
```

4.2.9 expanding_rank

pyg.timeseries._rank.**expanding_rank**(*a*, *axis=0*)

returns a rank of the current value within history, scaled to be -1 if it is the smallest and +1 if it is the largest - works on mumpy arrays too - skips nan, no ffill

Example

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series([1.,2., np.nan, 0.,4.,2.], drange(-5))
>>> rank = expanding_rank(a)
>>> assert eq(rank, pd.Series([0, 1, np.nan, -1, 1, 0.25], drange(-5)))
>>> #
>>> # 2 is largest in [1,2] so goes to 1;
>>> # 0 is smallest in [1,2,0] so goes to -1 etc.
```

Example numpy equivalent

```
>>> assert eq(expanding_rank(a.values), expanding_rank(a).values)
```

4.2.10 cumsum

pyg.timeseries._expanding.**cumsum**(*a*, *axis=0*, *data=None*, *state=None*)

equivalent to pandas a.expanding().sum().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

axis [int, optional] 0/1/-1. The default is 0.

t0,t1,data: state parameters to instantiate the calculation. t0 = total points so far, t2 = total(a) so far

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.expanding().sum(); ts = expanding_sum(a)
>>> assert eq(ts,panda)
```

Example nan handling

Unlike pandas, timeseries does not forward fill the nans.

```
>>> a[a<0.1] = np.nan
>>> panda = a.expanding().sum(); ts = expanding_sum(a)
```

```
>>> pd.concat([panda,ts], axis=1)
>>>                     0          1
>>> 1993-09-23          NaN          NaN
>>> 1993-09-24          NaN          NaN
>>> 1993-09-25     0.645944     0.645944
>>> 1993-09-26     2.816321     2.816321
>>> 1993-09-27     2.816321          NaN
>>>                 ...          ...
>>> 2021-02-03  3976.911348  3976.911348
>>> 2021-02-04  3976.911348          NaN
>>> 2021-02-05  3976.911348          NaN
>>> 2021-02-06  3976.911348          NaN
>>> 2021-02-07  3976.911348          NaN
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = expanding_sum(a)
>>> old_ts = expanding_sum_(old)
>>> new_ts = expanding_sum(new, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(expanding_sum(dict(x = a, y = a**2)), dict(x = expanding_sum(a), y␣
↪= expanding_sum(a**2)))
>>> assert eq(expanding_sum([a,a**2]), [expanding_sum(a), expanding_sum(a**2)])
```

4.2.11 cumprod

`pyg.timeseries._expanding.`**`cumprod`**`(`*a*, *axis=0*, *data=None*, *state=None*`)`

4.3 rolling window functions

4.3.1 rolling_mean

pyg.timeseries._rolling.**rolling_mean**(*a*, *n*, *axis=0*, *data=None*, *state=None*)

equivalent to pandas a.rolling(n).mean().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

n: int size of rolling window

axis [int, optional] 0/1/-1. The default is 0.

t0,t1,t2,data: state parameters to instantiate the calculation. t0,t1,t2= total(points),total(a),total(a**2) so far

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.rolling(10).mean(); ts = rolling_mean(a,10)
>>> assert abs(ts-panda).max()<1e-10
```

Example nan handling

Unlike pandas, timeseries does not include the nans in the rolling calculation: it skips them. Since pandas rolling engine does not skip nans, they propagate. In fact, having removed half the data points, rolling(10) will return 99% of nans

```
>>> a[a<0.1] = np.nan
>>> panda = a.rolling(10).mean(); ts = rolling_mean(a,10)
>>> print('#original:', len(nona(a)), 'timeseries:', len(nona(ts)), 'panda:',␣
↪len(nona(panda)), 'data points')
>>> #original: 4534 timeseries: 4525 panda: 6 data points
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = rolling_mean(a,10)
>>> old_ts = rolling_mean_(old,10)
>>> new_ts = rolling_mean(new, 10, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(rolling_mean(dict(x = a, y = a**2),10), dict(x = rolling_mean(a,10),
↪ y = rolling_mean(a**2,10)))
>>> assert eq(rolling_mean([a,a**2],10), [rolling_mean(a,10), rolling_mean(a**2,
↪10)])
```

4.3.2 rolling_rms

pyg.timeseries._rolling.**rolling_rms**(*a, n, axis=0, data=None, state=None*)

equivalent to pandas (a**2).rolling(n).mean()**0.5.

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

n: int size of rolling window

axis [int, optional] 0/1/-1. The default is 0.

t0,t1,t2,data: state parameters to instantiate the calculation. t0,t1,t2= total(points),total(a),total(a**2) so far

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = (a**2).rolling(10).mean()**0.5; ts = rolling_rms(a,10)
>>> assert abs(ts-panda).max()<1e-10
```

Example nan handling

Unlike pandas, timeseries does not include the nans in the rolling calculation: it skips them. Since pandas rolling engine does not skip nans, they propagate. In fact, having removed half the data points, rolling(10) will return 99% of nans

```
>>> a[a<0.1] = np.nan
>>> panda = (a**2).rolling(10).mean()**0.5; ts = rolling_rms(a,10)
>>> print('#original:', len(nona(a)), 'timeseries:', len(nona(ts)), 'panda:',␣
↪len(nona(panda)), 'data points')
>>> #original: 4534 timeseries: 4525 panda: 6 data points
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = rolling_rms(a,10)
>>> old_ts = rolling_rms_(old,10)
>>> new_ts = rolling_rms(new, 10, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(rolling_rms(dict(x = a, y = a**2),10), dict(x = rolling_rms(a,10),␣
↪y = rolling_rms(a**2,10)))
>>> assert eq(rolling_rms([a,a**2],10), [rolling_rms(a,10), rolling_rms(a**2,10)])
```

4.3.3 rolling_std

pyg.timeseries._rolling.**rolling_std**(*a*, *n*, *axis=0*, *data=None*, *state=None*)

equivalent to pandas a.rolling(n).std().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

n: int size of rolling window

axis [int, optional] 0/1/-1. The default is 0.

t0,t1,t2,data: state parameters to instantiate the calculation. t0,t1,t2= total(points),total(a),total(a**2) so far

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.rolling(10).std(); ts = rolling_std(a,10)
>>> assert abs(ts-panda).max()<1e-10
```

Example nan handling

Unlike pandas, timeseries does not include the nans in the rolling calculation: it skips them. Since pandas rolling engine does not skip nans, they propagate. In fact, having removed half the data points, rolling(10) will return 99.9% nans

```
>>> a[a<0.1] = np.nan
>>> panda = a.rolling(10).std(); ts = rolling_std(a,10)
>>> print('#original:', len(nona(a)), 'timeseries:', len(nona(ts)), 'panda:',␣
↪len(nona(panda)), 'data points')
>>> #original: 4534 timeseries: 4525 panda: 2 data points
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = rolling_std(a,10)
>>> old_ts = rolling_std_(old,10)
>>> new_ts = rolling_std(new, 10, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(rolling_std(dict(x = a, y = a**2),10), dict(x = rolling_std(a,10),␣
↪y = rolling_std(a**2,10)))
>>> assert eq(rolling_std([a,a**2],10), [rolling_std(a,10), rolling_std(a**2,10)])
```

4.3.4 rolling_sum

pyg.timeseries._rolling.**rolling_sum**(*a*, *n*, *axis=0*, *data=None*, *state=None*)
equivalent to pandas a.rolling(n).sum().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

n: int size of rolling window

axis [int, optional] 0/1/-1. The default is 0.

t0,t1,t2,data: state parameters to instantiate the calculation. t0,t1,t2= total(points),total(a),total(a**2) so far

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.rolling(10).sum(); ts = rolling_sum(a,10)
>>> assert abs(ts-panda).max()<1e-10
```

Example nan handling

Unlike pandas, timeseries does not include the nans in the rolling calculation: it skips them. Since pandas rolling engine does not skip nans, they propagate. In fact, having removed half the data points, rolling(10) will return 99.9% nans

```
>>> a[a<0.1] = np.nan
>>> panda = a.rolling(10).sum(); ts = rolling_sum(a,10)
>>> print('#original:', len(nona(a)), 'timeseries:', len(nona(ts)), 'panda:',␣
↪len(nona(panda)), 'data points')
>>> #original: 4534 timeseries: 4525 panda: 2 data points
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = rolling_sum(a,10)
>>> old_ts = rolling_sum_(old,10)
>>> new_ts = rolling_sum(new, 10, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(rolling_sum(dict(x = a, y = a**2),10), dict(x = rolling_sum(a,10),␣
↪y = rolling_sum(a**2,10)))
>>> assert eq(rolling_sum([a,a**2],10), [rolling_sum(a,10), rolling_sum(a**2,10)])
```

4.3.5 rolling_skew

pyg.timeseries._rolling.**rolling_skew**(*a*, *n*, *bias=False*, *axis=0*, *data=None*, *state=None*)

equivalent to pandas a.rolling(n).skew().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

n: int size of rolling window

bias: affects the skew calculation definition, see scipy documentation for details.

axis [int, optional] 0/1/-1. The default is 0.

t0,t1,t2,t3,vec,i,data: state parameters to instantiate the calculation. t0,t1,t2= total(points),total(a),total(a**2) so far

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.rolling(10).skew(); ts = rolling_skew(a,10)
>>> assert abs(ts-panda).max()<1e-10
```

Example nan handling

Unlike pandas, timeseries does not include the nans in the rolling calculation: it skips them. Since pandas rolling engine does not skip nans, they propagate. In fact, having removed half the data points, rolling(10) will return 99.9% nans

```
>>> a[a<0.1] = np.nan
>>> panda = a.rolling(10).skew(); ts = rolling_skew(a,10)
>>> print('#original:', len(nona(a)), 'timeseries:', len(nona(ts)), 'panda:',␣
↪len(nona(panda)), 'data points')
>>> #original: 4534 timeseries: 4525 panda: 2 data points
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = rolling_skew(a,10)
>>> old_ts = rolling_skew_(old,10)
>>> new_ts = rolling_skew(new, 10, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(rolling_skew(dict(x = a, y = a**2),10), dict(x = rolling_skew(a,10),
↪ y = rolling_skew(a**2,10)))
>>> assert eq(rolling_skew([a,a**2],10), [rolling_skew(a,10), rolling_skew(a**2,
↪10)])
```

4.3.6 rolling_min

pyg.timeseries._min.**rolling_min**(*a*, *n*, *axis=0*, *data=None*, *state=None*)

equivalent to pandas a.rolling(n).min().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

n: int size of rolling window

axis [int, optional] 0/1/-1. The default is 0.

vec,data: state parameters to instantiate the calculation. vec = recent history

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.rolling(10).min(); ts = rolling_min(a,10)
>>> assert abs(ts-panda).min()<1e-10
```

Example nan handling

Unlike pandas, timeseries does not include the nans in the rolling calculation: it skips them. Since pandas rolling engine does not skip nans, they propagate. In fact, having removed half the data points, rolling(10) will return 99% of nans

```
>>> a[a<0.1] = np.nan
>>> panda = a.rolling(10).min(); ts = rolling_min(a,10)
>>> print('#original:', len(nona(a)), 'timeseries:', len(nona(ts)), 'panda:',␣
↪len(nona(panda)), 'data points')
>>> #original: 4534 timeseries: 4525 panda: 6 data points
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = rolling_min(a,10)
>>> old_ts = rolling_min_(old,10)
>>> new_ts = rolling_min(new, 10, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(rolling_min(dict(x = a, y = a**2),10), dict(x = rolling_min(a,10),␣
↪y = rolling_min(a**2,10)))
>>> assert eq(rolling_min([a,a**2],10), [rolling_min(a,10), rolling_min(a**2,10)])
```

4.3.7 rolling_max

pyg.timeseries._max.**rolling_max**(*a*, *n*, *axis=0*, *data=None*, *state=None*)

equivalent to pandas a.rolling(n).max().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

n: int size of rolling window

axis [int, optional] 0/1/-1. The default is 0.

vec,data: state parameters to instantiate the calculation. vec = recent history

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.rolling(10).max(); ts = rolling_max(a,10)
>>> assert abs(ts-panda).max()<1e-10
```

Example nan handling

Unlike pandas, timeseries does not include the nans in the rolling calculation: it skips them. Since pandas rolling engine does not skip nans, they propagate. In fact, having removed half the data points, rolling(10) will return 99% of nans

```
>>> a[a<0.1] = np.nan
>>> panda = a.rolling(10).max(); ts = rolling_max(a,10)
>>> print('#original:', len(nona(a)), 'timeseries:', len(nona(ts)), 'panda:',␣
↪len(nona(panda)), 'data points')
>>> #original: 4534 timeseries: 4525 panda: 6 data points
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = rolling_max(a,10)
>>> old_ts = rolling_max_(old,10)
>>> new_ts = rolling_max(new, 10, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(rolling_max(dict(x = a, y = a**2),10), dict(x = rolling_max(a,10),
↪y = rolling_max(a**2,10)))
>>> assert eq(rolling_max([a,a**2],10), [rolling_max(a,10), rolling_max(a**2,10)])
```

4.3.8 rolling_median

`pyg.timeseries._median.`**`rolling_median`**(*a*, *n*, *axis=0*, *data=None*, *state=None*)

equivalent to pandas a.rolling(n).median().

- works with np.arrays
- handles nan without forward filling.
- supports state parameters

Parameters

a [array, pd.Series, pd.DataFrame or list/dict of these] timeseries

n: int size of rolling window

axis [int, optional] 0/1/-1. The default is 0.

vec,data: state parameters to instantiate the calculation. vec = recent history

Example agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> panda = a.rolling(10).median(); ts = rolling_median(a,10)
>>> assert abs(ts-panda).max()<1e-10
```

Example nan handling

Unlike pandas, timeseries does not include the nans in the rolling calculation: it skips them. Since pandas rolling engine does not skip nans, they propagate. In fact, having removed half the data points, rolling(10) will return 99% of nans

```
>>> a[a<0.1] = np.nan
>>> panda = a.rolling(10).median(); ts = rolling_median(a,10)
>>> print('#original:', len(nona(a)), 'timeseries:', len(nona(ts)), 'panda:',
↪len(nona(panda)), 'data points')
#original: 4634 timeseries: 4625 panda: 4 data points
```

Example state management

One can split the calculation and run old and new data separately.

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> ts = rolling_median(a,10)
>>> old_ts = rolling_median_(old,10)
>>> new_ts = rolling_median(new, 10, **old_ts)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example dict/list inputs

```
>>> assert eq(rolling_median(dict(x = a, y = a**2),10), dict(x = rolling_median(a,
↪10), y = rolling_median(a**2,10)))
>>> assert eq(rolling_median([a,a**2],10), [rolling_median(a,10), rolling_
↪median(a**2,10)])
```

4.3.9 rolling_quantile

pyg.timeseries._stride.**rolling_quantile**(*a*, *n*, *quantile=0.5*, *axis=0*, *data=None*, *state=None*)

equivalent to a.rolling(n).quantile(q) except… - supports numpy arrays - supports multiple q values

Example

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> res = rolling_quantile(a, 100, 0.3)
>>> assert sub_(res, a.rolling(100).quantile(0.3)).max() < 1e-13
```

Example multiple quantiles

```
>>> res = rolling_quantile(a, 100, [0.3, 0.5, 0.75])
>>> assert abs(res[0.3] - a.rolling(100).quantile(0.3)).max() < 1e-13
```

Example state management

```
>>> res = rolling_quantile(a, 100, 0.3)
>>> old = rolling_quantile_(a.iloc[:2000], 100, 0.3)
>>> new = rolling_quantile(a.iloc[2000:], 100, 0.3, **old)
>>> both = pd.concat([old.data, new])
>>> assert eq(both, res)
```

Parameters

a : array/timeseries n : integer

window size.

q [float or list of floats in [0,1]] quantile(s).

Returns

timeseries/array of quantile(s)

4.3.10 rolling_rank

pyg.timeseries._rank.**rolling_rank**(*a*, *n*, *axis=0*, *data=None*, *state=None*)

returns a rank of the current value within a given window, scaled to be -1 if it is the smallest and +1 if it is the largest - works on mumpy arrays too - skips nan, no ffill

Example

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = pd.Series([1.,2., np.nan, 0., 4., 2., 3., 1., 2.], drange(-8))
>>> rank = rolling_rank(a, 3)
>>> assert eq(rank.values, np.array([np.nan, np.nan, np.nan, -1, 1, 0, 0, -1, 0]))
>>> # 0 is smallest in [1,2,0] so goes to -1
>>> # 4 is largest in [2,0,4] so goes to +1
>>> # 2 is middle of [0,4,2] so goes to 0
```

Example numpy equivalent

```
>>> assert eq(rolling_rank(a.values, 10), rolling_rank(a, 10).values)
```

Example state management

```
>>> a = np.random.normal(0,1,10000)
>>> old = rolling_rank_(a[:5000], 10) # grab both data and state
>>> new = rolling_rank(a[5000:], 10, **old)
>>> assert eq(np.concatenate([old.data,new]), rolling_rank(a, 10))
```

4.4 exponentially weighted moving functions

4.4.1 ewma

pyg.timeseries._ewm.**ewma**(*a*, *n*, *time=None*, *axis=0*, *data=None*, *state=None*)

ewma is equivalent to a.ewm(n).mean() but with… - supports np.ndarrays as well as timeseries - handles nan by skipping them - allows state-management - ability to supply a 'clock' to the calculation

Parameters

a : array/timeseries n : int/fraction

The number or days (or a ratio) to scale the history

time [Calendar, 'b/d/y/m' or a timeseries of time (use clock(a) to see output)]

If time parameter is provided, we allow multiple observations per unit of time. i.e., converging to the last observation

- if we have intraday data, and set time = 'd', then
- the ewm calculation on last observations per day is what is retained.
- the ewm calculation on each intraday observation is same as an ewm(past EOD + current intraday observation)

state: dict, optional state parameters and are used to instantiate what the current average is based on history.

Example matching pandas

```
>>> import pandas as pd; import numpy as np; from pyg import *
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> ts = ewma(a,10); df = a.ewm(10).mean()
>>> assert abs(ts-df).max()<1e-10
```

Example numpy arrays support

```
>>> assert eq(ewma(a.values, 10), ewma(a,10).values)
```

Example nan handling

```
>>> a[a.values<0.1] = np.nan
>>> ts = ewma(a,10, time = 'i'); df = a.ewm(10).mean() # note: pandas assumes,
↪'time' pass per index entry, even if value is nan
>>> assert abs(ts-df).max()<1e-10
```

```
>>> pd.concat([ts,df], axis=1)
>>>                    0         1
>>> 1993-09-24  0.263875  0.263875
>>> 1993-09-25       NaN  0.263875
>>> 1993-09-26       NaN  0.263875
>>> 1993-09-27       NaN  0.263875
>>> 1993-09-28       NaN  0.263875
>>>                  ...       ...
>>> 2021-02-04       NaN  0.786506
>>> 2021-02-05  0.928817  0.928817
>>> 2021-02-06       NaN  0.928817
>>> 2021-02-07  0.839168  0.839168
>>> 2021-02-08  0.831109  0.831109
```

Example state management

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> old_ts = ewma_(old, 10)
>>> new_ts = ewma(new, 10, **old_ts) # instantiation with previous ewma
>>> ts = ewma(a,10)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example Support for time & clock

```
>>> daily = a
>>> monthly = daily.resample('M').last()
>>> m_ts = ewma(monthly, 3) ## 3-month ewma run on monthly data
>>> d_ts = ewma(daily, 3, 'm') ## 3-month ewma run on daily data
>>> daily_resampled_to_month = d_ts.resample('M').last()
>>> assert abs(daily_resampled_to_month - m_ts).max() < 1e-10
```

So you can run a 3-monthly ewma on daily, where within month, most recent value is used with the EOM history.

Example Support for dict/list of arrays

```
>>> x = pd.Series(np.random.normal(0,1,1000), drange(-999)); y = pd.Series(np.
↪random.normal(0,1,1000), drange(-999))
>>> a = dict(x = x, y = y)
>>> assert eq(ewma(dict(x=x, y=y),10), dict(x=ewma(x,10), y=ewma(y,10)))
>>> assert eq(ewma([x,y],10), [ewma(x,10), ewma(y,10)])
```

Returns

an array/timeseries of ewma

4.4.2 ewmrms

pyg.timeseries._ewm.**ewmrms**(*a*, *n*, *time=None*, *axis=0*, *data=None*, *state=None*)

ewmrms is equivalent to (a**2).ewm(n).mean()**0.5 but with... - supports np.ndarrays as well as timeseries - handles nan by skipping them - allows state-management - ability to supply a 'clock' to the calculation

Parameters

a : array/timeseries n : int/fraction

The number or days (or a ratio) to scale the history

time [Calendar, 'b/d/y/m' or a timeseries of time (use clock(a) to see output)]

If time parameter is provided, we allow multiple observations per unit of time. i.e., converging to the last observation

- if we have intraday data, and set time = 'd', then
- the ewm calculation on last observations per day is what is retained.
- the ewm calculation on each intraday observation is same as an ewm(past EOD + current intraday observation)

state: dict, optional state parameters and are used to instantiate what the current average is based on history.

Example matching pandas

```
>>> import pandas as pd; import numpy as np; from pyg import *
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> ts = ewmrms(a,10); df = (a**2).ewm(10).mean()**0.5
>>> assert abs(ts-df).max()<1e-10
```

Example numpy arrays support

```
>>> assert eq(ewmrms(a.values, 10), ewmrms(a,10).values)
```

Example nan handling

```
>>> a[a.values<0.1] = np.nan
>>> ts = ewmrms(a,10, time = 'i'); df = (a**2).ewm(10).mean()**0.5 # note: pandas␣
↪assumes, 'time' pass per index entry, even if value is nan
>>> assert abs(ts-df).max()<1e-10
```

```
>>> pd.concat([ts,df], axis=1)
>>>                        0         1
>>> 1993-09-24  0.263875  0.263875
>>> 1993-09-25       NaN  0.263875
>>> 1993-09-26       NaN  0.263875
>>> 1993-09-27       NaN  0.263875
>>> 1993-09-28       NaN  0.263875
>>>                  ...       ...
>>> 2021-02-04       NaN  0.786506
>>> 2021-02-05  0.928817  0.928817
>>> 2021-02-06       NaN  0.928817
>>> 2021-02-07  0.839168  0.839168
>>> 2021-02-08  0.831109  0.831109
```

Example state management

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> old_ts = ewmrms_(old, 10)
>>> new_ts = ewmrms(new, 10, **old_ts) # instantiation with previous ewma
>>> ts = ewmrms(a,10)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example Support for time & clock

```
>>> daily = a
>>> monthly = daily.resample('M').last()
>>> m_ts = ewmrms(monthly, 3) ## 3-month ewma run on monthly data
>>> d_ts = ewmrms(daily, 3, 'm') ## 3-month ewma run on daily data
>>> daily_resampled_to_month = d_ts.resample('M').last()
>>> assert abs(daily_resampled_to_month - m_ts).max() < 1e-10
```

So you can run a 3-monthly ewma on daily, where within month, most recent value is used with the EOM history.

Example Support for dict/list of arrays

```
>>> x = pd.Series(np.random.normal(0,1,1000), drange(-999)); y = pd.Series(np.
↪random.normal(0,1,1000), drange(-999))
>>> a = dict(x = x, y = y)
>>> assert eq(ewmrms(dict(x=x, y=y),10), dict(x=ewmrms(x,10), y=ewmrms(y,10)))
>>> assert eq(ewmrms([x,y],10), [ewmrms(x,10), ewmrms(y,10)])
```

Returns

an array/timeseries of ewma

4.4.3 ewmstd

pyg.timeseries._ewm.**ewmstd**(*a*, *n*, *time=None*, *min_sample=0.25*, *bias=False*, *axis=0*, *data=None*, *state=None*)

ewmstd is equivalent to a.ewm(n).std() but with… - supports np.ndarrays as well as timeseries - handles nan by skipping them - allows state-management - ability to supply a 'clock' to the calculation

Parameters

a : array/timeseries n : int/fraction

The number or days (or a ratio) to scale the history

time [Calendar, 'b/d/y/m' or a timeseries of time (use clock(a) to see output)]

If time parameter is provided, we allow multiple observations per unit of time. i.e., converging to the last observation

- if we have intraday data, and set time = 'd', then
- the ewm calculation on last observations per day is what is retained.
- the ewm calculation on each intraday observation is same as an ewm(past EOD + current intraday observation)

state: dict, optional state parameters and are used to instantiate what the current average is based on history.

Example matching pandas

```
>>> import pandas as pd; import numpy as np; from pyg import *
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> ts = ewmstd(a,10); df = a.ewm(10).std()
>>> assert abs(ts-df).max()<1e-10
>>> ts = ewmstd(a,10, bias = True); df = a.ewm(10).std(bias = True)
>>> assert abs(ts-df).max()<1e-10
```

Example numpy arrays support

```
>>> assert eq(ewmstd(a.values, 10), ewmstd(a,10).values)
```

Example nan handling

```
>>> a[a.values<-0.1] = np.nan
>>> ts = ewmstd(a,10, time = 'i'); df = a.ewm(10).std() # note: pandas assumes,
↪'time' pass per index entry, even if value is nan
>>> assert abs(ts-df).max()<1e-10
>>> ts = ewmstd(a,10, time = 'i', bias = True); df = a.ewm(10).std(bias = True) #␣
↪note: pandas assumes, 'time' pass per index entry, even if value is nan
>>> assert abs(ts-df).max()<1e-10
```

Example state management

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> old_ts = ewmstd_(old, 10)
>>> new_ts = ewmstd(new, 10, **old_ts) # instantiation with previous ewma
```

(continues on next page)

(continued from previous page)

```
>>> ts = ewmstd(a,10)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example Support for time & clock

```
>>> daily = a
>>> monthly = daily.resample('M').last()
>>> m_ts = ewmstd(monthly, 3) ## 3-month ewma run on monthly data
>>> d_ts = ewmstd(daily, 3, 'm') ## 3-month ewma run on daily data
>>> daily_resampled_to_month = d_ts.resample('M').last()
>>> assert abs(daily_resampled_to_month - m_ts).max() < 1e-10
```

So you can run a 3-monthly ewma on daily, where within month, most recent value is used with the EOM history.

Example Support for dict/list of arrays

```
>>> x = pd.Series(np.random.normal(0,1,1000), drange(-999)); y = pd.Series(np.
↪random.normal(0,1,1000), drange(-999))
>>> a = dict(x = x, y = y)
>>> assert eq(ewmstd(dict(x=x, y=y),10), dict(x=ewmstd(x,10), y=ewmstd(y,10)))
>>> assert eq(ewmstd([x,y],10), [ewmstd(x,10), ewmstd(y,10)])
```

Returns

an array/timeseries of ewma

4.4.4 ewmvar

pyg.timeseries._ewm.**ewmvar**(*a*, *n*, *time=None*, *min_sample=0.25*, *bias=False*, *axis=0*, *data=None*, *state=None*)

ewmstd is equivalent to a.ewm(n).var() but with... - supports np.ndarrays as well as timeseries - handles nan by skipping them - allows state-management - ability to supply a 'clock' to the calculation

Parameters

a : array/timeseries n : int/fraction

The number or days (or a ratio) to scale the history

time [Calendar, 'b/d/y/m' or a timeseries of time (use clock(a) to see output)]

If time parameter is provided, we allow multiple observations per unit of time. i.e., converging to the last observation

- if we have intraday data, and set time = 'd', then
- the ewm calculation on last observations per day is what is retained.
- the ewm calculation on each intraday observation is same as an ewm(past EOD + current intraday observation)

state: dict, optional state parameters and are used to instantiate what the current average is based on history.

Example matching pandas

```
>>> import pandas as pd; import numpy as np; from pyg import *
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> ts = ewmvar(a,10); df = a.ewm(10).var()
>>> assert abs(ts-df).max()<1e-10
>>> ts = ewmvar(a,10, bias = True); df = a.ewm(10).var(bias = True)
>>> assert abs(ts-df).max()<1e-10
```

Example numpy arrays support

```
>>> assert eq(ewmvar(a.values, 10), ewmvar(a,10).values)
```

Example nan handling

```
>>> a[a.values<-0.1] = np.nan
>>> ts = ewmvar(a,10, time = 'i'); df = a.ewm(10).var() # note: pandas assumes,
↪'time' pass per index entry, even if value is nan
>>> assert abs(ts-df).max()<1e-10
>>> ts = ewmvar(a,10, time = 'i', bias = True); df = a.ewm(10).var(bias = True) #␣
↪note: pandas assumes, 'time' pass per index entry, even if value is nan
>>> assert abs(ts-df).max()<1e-10
```

Example state management

```
>>> old = a.iloc[:5000]
>>> new = a.iloc[5000:]
>>> old_ts = ewmvar_(old, 10)
>>> new_ts = ewmvar(new, 10, **old_ts) # instantiation with previous ewma
>>> ts = ewmvar(a,10)
>>> assert eq(new_ts, ts.iloc[5000:])
```

Example Support for time & clock

```
>>> daily = a
>>> monthly = daily.resample('M').last()
>>> m_ts = ewmvar(monthly, 3) ## 3-month ewma run on monthly data
>>> d_ts = ewmvar(daily, 3, 'm') ## 3-month ewma run on daily data
>>> daily_resampled_to_month = d_ts.resample('M').last()
>>> assert abs(daily_resampled_to_month - m_ts).max() < 1e-10
```

So you can run a 3-monthly ewma on daily, where within month, most recent value is used with the EOM history.

Example Support for dict/list of arrays

```
>>> x = pd.Series(np.random.normal(0,1,1000), drange(-999)); y = pd.Series(np.
↪random.normal(0,1,1000), drange(-999))
>>> a = dict(x = x, y = y)
>>> assert eq(ewmvar(dict(x=x, y=y),10), dict(x=ewmvar(x,10), y=ewmvar(y,10)))
>>> assert eq(ewmvar([x,y],10), [ewmvar(x,10), ewmvar(y,10)])
```

Returns

an array/timeseries of ewma

4.4.5 ewmcor

pyg.timeseries._ewm.**ewmcor**(*a*, *b*, *n*, *time=None*, *min_sample=0.25*, *bias=True*, *axis=0*, *data=None*, *state=None*)

calculates pair-wise correlation between a and b.

a : array/timeseries b : array/timeseries n : int/fraction

The number or days (or a ratio) to scale the history

time [Calendar, 'b/d/y/m' or a timeseries of time (use clock(a) to see output)]

If time parameter is provided, we allow multiple observations per unit of time. i.e., converging to the last observation

- if we have intraday data, and set time = 'd', then
- the ewm calculation on last observations per day is what is retained.
- the ewm calculation on each intraday observation is same as an ewm(past EOD + current intraday observation)

min_sample [floar, optional] minimum weight of observations before we return a reading. The default is 0.25. This ensures that we don't get silly numbers due to small population.

bias [book, optional] vol estimation for a and b should really by unbiased. Nevertheless, we track pandas and set bias = True as a default.

axis [int, optional] axis of calculation. The default is 0.

data [place holder, ignore, optional] ignore. The default is None.

state [dict, optional] Output from a previous run of *ewmcor*. The default is None.

Example matching pandas

```
>>> import pandas as pd; import numpy as np; from pyg import *
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> b = pd.Series(np.random.normal(0,1,9000), drange(-8999))
>>> ts = ewmcor(a, b, n = 10); df = a.ewm(10).corr(b)
>>> assert abs(ts-df).max()<1e-10
```

Example numpy arrays support

```
>>> assert eq(ewmcor(a.values, b.values, 10), ewmcor(a, b, 10).values)
```

Example nan handling

```
>>> a[a.values<-0.1] = np.nan
>>> ts = ewmcor(a, b, 10, time = 'i'); df = a.ewm(10).corr(b) # note: pandas␣
↪assumes, 'time' pass per index entry, even if value is nan
>>> assert abs(ts-df).max()<1e-10
```

Example state management

```
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> b = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> old_a = a.iloc[:5000]; old_b = b.iloc[:5000]
>>> new_a = a.iloc[5000:]; new_b = b.iloc[5000:]
>>> old_ts = ewmcor_(old_a, old_b, 10)
>>> new_ts = ewmcor(new_a, new_b, 10, **old_ts) # instantiation with previous ewma
>>> ts = ewmcor(a,b,10)
>>> assert eq(new_ts, ts.iloc[5000:])
```

4.4.6 ewmskew

`pyg.timeseries._ewm.`**`ewmskew`**(*a*, *n*, *time=None*, *bias=False*, *min_sample=0.25*, *axis=0*, *data=None*, *state=None*)

Equivalent to a.ewm(n).skew() but with… - supports np.ndarrays as well as timeseries - handles nan by skipping them - allows state-management - ability to supply a 'clock' to the calculation

Parameters

a : array/timeseries n : int/fraction

The number or days (or a ratio) to scale the history

time [Calendar, 'b/d/y/m' or a timeseries of time (use clock(a) to see output)]

If time parameter is provided, we allow multiple observations per unit of time. i.e., converging to the last observation

- if we have intraday data, and set time = 'd', then
- the ewm calculation on last observations per day is what is retained.
- the ewm calculation on each intraday observation is same as an ewm(past EOD + current intraday observation)

state: dict, optional state parameters and are used to instantiate what the current average is based on history.

Example matching pandas

```
>>> import pandas as pd; import numpy as np; from pyg import *
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> x = a.ewm(10).skew()
```

```
>>> old = a.iloc[:10]
>>> new = a.iloc[10:]
f = ewmskew_
for f in [ewma_, ewmstd_, ewmrms_, ewmskew_, ]:
    both = f(a, 3)
    o = f(old, 3)
    n = f(new, 3, **o)
    assert eq(o.data, both.data.iloc[:10])
    assert eq(n.data, both.data.iloc[10:])
    assert both - 'data' == n - 'data'
```

```
>>> assert abs(a.ewm(10).mean() - ewma(a,10)).max() < 1e-14
>>> assert abs(a.ewm(10).std() - ewmstd(a,10)).max() < 1e-14
```

Example numpy arrays support

```
>>> assert eq(ewma(a.values, 10), ewma(a,10).values)
```

Example nan handling

while panadas ffill values, timeseries skips nans:

```
>>> a = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> a[a.values>0.1] = np.nan
>>> ts = ewma(a,10)
>>> assert eq(ts[~np.isnan(ts)], ewma(a[~np.isnan(a)], 10))
```

Example initiating the ewma with past state

```
>>> old = np.random.normal(0,1,100)
>>> new = np.random.normal(0,1,100)
>>> old_ = ewma_(old, 10)
>>> new_ = ewma(new, 10, t0 = old_ewma.t0, t1 = old_ewma.t1) # instantiation with␣
↪previous ewma
>>> new_2 = ewma(np.concatenate([old,new]), 10)[-100:]
>>> assert eq(new_ewma, new_ewma2)
```

Example Support for time & clock

```
>>> daily = pd.Series(np.random.normal(0,1,10000), drange(-9999)).cumsum()
>>> monthly = daily.resample('M').last()
>>> m = ewma(monthly, 3) ## 3-month ewma run on monthly data
>>> d = ewma(daily, 3, 'm') ## 3-month ewma run on daily data
>>> daily_resampled_to_month = d.resample('M').last()
>>> assert abs(daily_resampled_to_month - m).max() < 1e-10
```

So you can run a 3-monthly ewma on daily, where within month, most recent value is used with the EOM history.

Returns

an array/timeseries of ewma

4.5 functions exposing their state

4.5.1 simple functions

`pyg.timeseries._rolling.`**`diff_`**(*a, n=1, axis=0, data=None, instate=None*)

`pyg.timeseries._rolling.`**`shift_`**(*a, n=1, axis=0, instate=None*)

`pyg.timeseries._rolling.`**`ratio_`**(*a, n=1, data=None, instate=None*)

`pyg.timeseries._ts.`**`ts_count_`**(*a, axis=0, data=None, instate=None*)
ts_count_(a) is equivalent to ts_count(a) except vec is also returned. See ts_count for full documentation

`pyg.timeseries._ts.`**`ts_sum_`**(*a, axis=0, data=None, instate=None*)
ts_sum_(a) is equivalent to ts_sum(a) except vec is also returned. See ts_sum for full documentation

`pyg.timeseries._ts.`**`ts_mean_`**(*a, axis=0, data=None, instate=None*)
ts_mean_(a) is equivalent to ts_mean(a) except vec is also returned. See ts_mean for full documentation

pyg.timeseries._ts.**ts_rms_**(*a*, *axis=0*, *data=None*, *instate=None*)
ts_rms_(a) is equivalent to ts_rms(a) except it also returns vec see ts_rms for full documentation

pyg.timeseries._ts.**ts_std_**(*a*, *axis=0*, *data=None*, *instate=None*)
ts_std_(a) is equivalent to ts_std(a) except vec is also returned. See ts_std for full documentation

pyg.timeseries._ts.**ts_skew_**(*a*, *bias=False*, *min_sample=0.25*, *axis=0*, *data=None*, *instate=None*)
ts_skew_(a) is equivalent to ts_skew except vec is also returned. See ts_skew for full details

pyg.timeseries._ts.**ts_max_**(*a*, *axis=0*, *data=None*, *instate=None*)
ts_max(a) is equivalent to pandas a.min()

pyg.timeseries._ts.**ts_max_**(*a*, *axis=0*, *data=None*, *instate=None*)
ts_max(a) is equivalent to pandas a.min()

pyg.timeseries._rolling.**ffill_**(*a*, *n=0*, *axis=0*, *instate=None*)
returns a forward filled array, up to n values forward. supports state manegement

4.5.2 expanding window functions

pyg.timeseries._expanding.**expanding_mean_**(*a*, *axis=0*, *data=None*, *instate=None*)
Equivalent to expanding_mean(a) but returns also the state variables t0. . . t1. For full documentation, look at expanding_mean.__doc__

pyg.timeseries._expanding.**expanding_rms_**(*a*, *axis=0*, *data=None*, *instate=None*)
Equivalent to expanding_rms(a) but returns also the state variables t0. . . t2. For full documentation, look at expanding_rms.__doc__

pyg.timeseries._expanding.**expanding_std_**(*a*, *axis=0*, *data=None*, *instate=None*)
Equivalent to expanding_std(a) but returns also the state variables t0. . . t2. For full documentation, look at expanding_std.__doc__

pyg.timeseries._expanding.**expanding_sum_**(*a*, *axis=0*, *data=None*, *instate=None*)
Equivalent to expanding_sum(a) but returns also the state variables t0. . . t1. For full documentation, look at expanding_sum.__doc__

pyg.timeseries._expanding.**expanding_skew_**(*a*, *bias=False*, *axis=0*, *data=None*, *instate=None*)
Equivalent to expanding_kew(a) but returns also the state variables t0. . . t3. For full documentation, look at expanding_skew.__doc__

pyg.timeseries._min.**expanding_min_**(*a*, *axis=0*, *data=None*, *instate=None*)
Equivalent to a.expanding().min() but returns the full state: i.e. both data: the expanding().min() m: the current minimum

pyg.timeseries._max.**expanding_max_**(*a*, *axis=0*, *data=None*, *instate=None*)
Equivalent to a.expanding().max() but returns the full state: i.e. both data: the expanding().max() m: the current maximum

pyg.timeseries._expanding.**cumsum_**(*a*, *axis=0*, *data=None*, *instate=None*)
Equivalent to expanding_sum(a) but returns also the state variables t0. . . t1. For full documentation, look at expanding_sum.__doc__

pyg.timeseries._expanding.**cumprod_**(*a*, *axis=0*, *data=None*, *instate=None*)

4.5.3 rolling window functions

`pyg.timeseries._rolling.`**`rolling_mean_`**(*a*, *n*, *axis=0*, *data=None*, *instate=None*)
Equivalent to rolling_mean(a) but returns also the state variables t0,t1 etc. For full documentation, look at rolling_mean.__doc__

`pyg.timeseries._rolling.`**`rolling_rms_`**(*a*, *n*, *axis=0*, *data=None*, *instate=None*)
Equivalent to rolling_rms(a) but returns also the state variables t0,t1 etc. For full documentation, look at rolling_rms.__doc__

`pyg.timeseries._rolling.`**`rolling_std_`**(*a*, *n*, *axis=0*, *data=None*, *instate=None*)
Equivalent to rolling_std(a) but returns also the state variables t0,t1 etc. For full documentation, look at rolling_std.__doc__

`pyg.timeseries._rolling.`**`rolling_sum_`**(*a*, *n*, *axis=0*, *data=None*, *instate=None*)
Equivalent to rolling_sum(a) but returns also the state variables t0,t1 etc. For full documentation, look at rolling_sum.__doc__

`pyg.timeseries._rolling.`**`rolling_skew_`**(*a*, *n*, *bias=False*, *axis=0*, *data=None*, *instate=None*)
Equivalent to rolling_skew(a) but returns also the state variables t0,t1 etc. For full documentation, look at rolling_skew.__doc__

`pyg.timeseries._min.`**`rolling_min_`**(*a*, *n*, *vec=None*, *axis=0*, *data=None*, *instate=None*)
Equivalent to rolling_min(a) but returns also the state. For full documentation, look at rolling_min.__doc__

`pyg.timeseries._max.`**`rolling_max_`**(*a*, *n*, *axis=0*, *data=None*, *instate=None*)
Equivalent to rolling_max(a) but returns also the state. For full documentation, look at rolling_max.__doc__

`pyg.timeseries._median.`**`rolling_median_`**(*a*, *n*, *axis=0*, *data=None*, *instate=None*)
Equivalent to rolling_median(a) but returns also the state. For full documentation, look at rolling_median.__doc__

`pyg.timeseries._rank.`**`rolling_rank_`**(*a*, *n*, *axis=0*, *data=None*, *instate=None*)
Equivalent to rolling_rank(a) but returns also the state variables. For full documentation, look at rolling_rank.__doc__

`pyg.timeseries._stride.`**`rolling_quantile_`**(*a*, *n*, *quantile=0.5*, *axis=0*, *data=None*, *instate=None*)
Equivalent to rolling_quantile(a) but returns also the state. For full documentation, look at rolling_quantile.__doc__

4.5.4 exponentially weighted moving functions

`pyg.timeseries._ewm.`**`ewma_`**(*a*, *n*, *time=None*, *data=None*, *instate=None*)
Equivalent to ewma but returns a state parameter for instantiation of later calculations. See ewma documentation for more details

`pyg.timeseries._ewm.`**`ewmrms_`**(*a*, *n*, *time=None*, *axis=0*, *data=None*, *instate=None*)
Equivalent to ewmrms but returns a state parameter for instantiation of later calculations. See ewmrms documentation for more details

`pyg.timeseries._ewm.`**`ewmstd_`**(*a*, *n*, *time=None*, *min_sample=0.25*, *bias=False*, *axis=0*, *data=None*, *instate=None*)
Equivalent to ewmstd but returns a state parameter for instantiation of later calculations. See ewmstd documentation for more details

`pyg.timeseries._ewm.`**`ewmvar_`**(*a*, *n*, *time=None*, *min_sample=0.25*, *bias=False*, *axis=0*, *data=None*, *instate=None*)
Equivalent to ewmvar but returns a state parameter for instantiation of later calculations. See ewmvar documentation for more details

pyg.timeseries._ewm.**ewmcor_**(*a*, *b*, *n*, *time=None*, *min_sample=0.25*, *bias=True*, *axis=0*, *data=None*, *instate=None*)
Equivalent to ewmcor but returns a state parameter for instantiation of later calculations. See ewmcor documentation for more details

pyg.timeseries._ewm.**ewmskew_**(*a*, *n*, *time=None*, *bias=False*, *min_sample=0.25*, *axis=0*, *data=None*, *instate=None*)
Equivalent to ewmskew but returns a state parameter for instantiation of later calculations. See ewmskew documentation for more details

4.6 Index handling

4.6.1 df_fillna

pyg.timeseries._index.**df_fillna**(*df*, *method=None*, *axis=0*, *limit=None*)
Equivelent to df.fillna() except:

- support np.ndarray as well as dataframes
- support multiple methods of filling/interpolation
- supports removal of nan from the start/all of the timeseries
- supports action on multiple timeseries

Parameters

df : dataframe/numpy array

method [string, list of strings or None, optional] Either a fill method (bfill, ffill, pad) Or an interplation method: 'linear', 'time', 'index', 'values', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'spline', 'polynomial', 'from_derivatives', 'piecewise_polynomial', 'pchip', 'akima', 'cubicspline' Or 'fnna': removes all to the first non nan Or 'nona': removes all nans

axis [int, optional] axis. The default is 0.

limit [TYPE, optional] when filling, how many nan get filled. The default is None (indefinite)

Example method ffill or bfill

```
>>> from pyg import *; import numpy as np
>>> df = np.array([np.nan, 1., np.nan, 9, np.nan, 25])
>>> assert eq(df_fillna(df, 'ffill'), np.array([ np.nan, 1.,  1.,  9.,  9., 25.]))
>>> assert eq(df_fillna(df, ['ffill','bfill']), np.array([ 1., 1.,  1.,  9.,  9.,␣
↪25.]))
>>> assert eq(df_fillna(df, ['ffill','bfill']), np.array([ 1., 1.,  1.,  9.,  9.,␣
↪25.]))
```

```
>>> df = np.array([np.nan, 1., np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, 9,␣
↪np.nan, 25])
>>> assert eq(df_fillna(df, 'ffill', limit = 2), np.array([np.nan,  1.,  1.,  1.,␣
↪np.nan, np.nan, np.nan, np.nan,  9.,  9., 25.]))
```

df_fillna does not maintain state of latest 'prev' value: use **ffill_** for that.

Example interpolation methods

```
>>> from pyg import *; import numpy as np
>>> df = np.array([np.nan, 1., np.nan, 9, np.nan, 25])
>>> assert eq(df_fillna(df, 'linear'), np.array([ np.nan, 1.,  5.,  9.,  17., 25.
↪]))
>>> assert eq(df_fillna(df, 'quadratic'), np.array([ np.nan, 1.,  4.,  9.,  16.,␣
↪25.]))
```

Example method = fnna and nona

```
>>> from pyg import *; import numpy as np
>>> ts = np.array([np.nan] * 10 + [1.] * 10 + [np.nan])
>>> assert eq(df_fillna(ts, 'fnna'), np.array([1.]*10 + [np.nan]))
>>> assert eq(df_fillna(ts, 'nona'), np.array([1.]*10))
```

```
>>> assert len(df_fillna(np.array([np.nan]), 'nona')) == 0
>>> assert len(df_fillna(np.array([np.nan]), 'fnna')) == 0
```

Returns

array/dataframe with nans removed/filled

4.6.2 df_index

pyg.timeseries._index.**df_index**(*seq*, *index='inner'*)

Determines a joint index of multiple timeseries objects.

Parameters

seq [sequence whose index needs to be determined] a (possible nested) sequence of timeseries/non-timeseries object within lists/dicts

index [str, optional] method to determine the index. The default is 'inner'.

Returns

pd.Index The joint index.

Example

```
>>> tss = [pd.Series(np.random.normal(0,1,10), drange(-i, 9-i)) for i in range(5)]
>>> more_tss_as_dict = dict(zip('abcde',[pd.Series(np.random.normal(0,1,10),␣
↪drange(-i, 9-i)) for i in range(5)]))
>>> res = df_index(tss + [more_tss_as_dict], 'inner')
>>> assert len(res) == 6
>>> res = df_index(more_tss_as_dict, 'outer')
>>> assert len(res) == 14
```

4.6.3 df_reindex

pyg.timeseries._index.**df_reindex**(*ts*, *index=None*, *method=None*, *limit=None*)

A slightly more general version of df.reindex(index)

Parameters

ts [dataframe or numpy array (or list/dict of theses)] timeseries to be reindexed

index [str, timeseries, pd.Index.] The new index

method [str, list of str, float, optional] various methods of handling nans are available. The default is None. See df_fillna for a full list.

Returns

timeseries/np.ndarray (or list/dict of theses) timeseries reindex.

Example index = inner/outer

```
>>> tss = [pd.Series(np.random.normal(0,1,10), drange(-i, 9-i)) for i in range(5)]
>>> res = df_reindex(tss, 'inner')
>>> assert len(res[0]) == 6
>>> res = df_reindex(tss, 'outer')
>>> assert len(res[0]) == 14
```

Example index provided

```
>>> tss = [pd.Series(np.random.normal(0,1,10), drange(-i, 9-i)) for i in range(5)]
>>> res = df_reindex(tss, tss[0])
>>> assert eq(res[0], tss[0])
>>> res = df_reindex(tss, tss[0].index)
>>> assert eq(res[0], tss[0])
```

4.6.4 presync

pyg.timeseries._index.**presync**()

Much of timeseries analysis in Pandas is spent aligning multiple timeseries before feeding them into a function. presync allows easy presynching of all paramters of a function.

Parameters

function [callable, optional] function to be presynched. The default is None.

index [str, optional] index join policy. The default is 'inner'.

method [str/int/list of these, optional] method of nan handling. The default is None.

columns [str, optional] columns join policy. The default is 'inner'.

default [float, optional] value when no data is available. The default is np.nan.

Returns

presynch-decorated function

Example

```
>>> from pyg import *
>>> x = pd.Series([1,2,3,4], drange(-3))
>>> y = pd.Series([1,2,3,4], drange(-4,-1))
>>> z = pd.DataFrame([[1,2],[3,4]], drange(-3,-2), ['a','b'])
>>> addition = lambda a, b: a+b
```

#We get some nonsensical results:

```
>>> assert list(addition(x,z).columns) ==  list(x.index) + ['a', 'b']
```

#But:

```
>>> assert list(presync(addition)(x,z).columns) == ['a', 'b']
>>> res = presync(addition, index='outer', method = 'ffill')(x,z)
>>> assert eq(res.a.values, np.array([2,5,6,7]))
```

Example 2 alignment works for parameters 'buried' within…

```
>>> function = lambda a, b: a['x'] + a['y'] + b
>>> f = presync(function, 'outer', method = 'ffill')
>>> res = f(dict(x = x, y = y), b = z)
>>> assert eq(res, pd.DataFrame(dict(a = [np.nan, 4, 8, 10, 11], b = [np.nan, 5,␣
↪9, 11, 12]), index = drange(-4)))
```

Example 3 alignment of numpy arrays

```
>>> addition = lambda a, b: a+b
>>> a = presync(addition)
>>> assert eq(a(pd.Series([1,2,3,4], drange(-3)), np.array([[1,2,3,4]]).T),  pd.
↪Series([2,4,6,8], drange(-3)))
>>> assert eq(a(pd.Series([1,2,3,4], drange(-3)), np.array([1,2,3,4])),  pd.
↪Series([2,4,6,8], drange(-3)))
>>> assert eq(a(pd.Series([1,2,3,4], drange(-3)), np.array([[1,2,3,4],[5,6,7,8]]).
↪T),  pd.DataFrame({0:[2,4,6,8], 1:[6,8,10,12]}, drange(-3)))
>>> assert eq(a(np.array([1,2,3,4]), np.array([[1,2,3,4]]).T),  np.array([2,4,6,
↪8]))
```

Example 4 inner join alignment of columns in dataframes by default

```
>>> x = pd.DataFrame({'a':[2,4,6,8], 'b':[6,8,10,12.]}, drange(-3))
>>> y = pd.DataFrame({'wrong':[2,4,6,8], 'columns':[6,8,10,12]}, drange(-3))
>>> assert len(a(x,y)) == 0
>>> y = pd.DataFrame({'a':[2,4,6,8], 'other':[6,8,10,12.]}, drange(-3))
>>> assert eq(a(x,y),x[['a']]*2)
>>> y = pd.DataFrame({'a':[2,4,6,8], 'b':[6,8,10,12.]}, drange(-3))
>>> assert eq(a(x,y),x*2)
>>> y = pd.DataFrame({'column name for a single column dataframe is ignored':[1,1,
↪1,1]}, drange(-3))
>>> assert eq(a(x,y),x+1)
```

```
>>> a = presync(addition, columns = 'outer')
>>> y = pd.DataFrame({'other':[2,4,6,8], 'a':[6,8,10,12]}, drange(-3))
>>> assert sorted(a(x,y).columns) == ['a','b','other']
```

Example 4 ffilling, bfilling

```
>>> x = pd.Series([1.,np.nan,3.,4.], drange(-3))
>>> y = pd.Series([1.,np.nan,3.,4.], drange(-4,-1))
>>> assert eq(a(x,y), pd.Series([np.nan, np.nan,7], drange(-3,-1)))
```

but, we provide easy conversion of internal parameters of presync:

```
>>> assert eq(a.ffill(x,y), pd.Series([2,4,7], drange(-3,-1)))
>>> assert eq(a.bfill(x,y), pd.Series([4,6,7], drange(-3,-1)))
>>> assert eq(a.oj(x,y), pd.Series([np.nan, np.nan, np.nan, 7, np.nan], drange(-
↪4)))
>>> assert eq(a.oj.ffill(x,y), pd.Series([np.nan, 2, 4, 7, 8], drange(-4)))
```

Example 5 indexing to a specific index

```
>>> index = pd.Index([dt(-3), dt(-1)])
>>> a = presync(addition, index = index)
>>> x = pd.Series([1.,np.nan,3.,4.], drange(-3))
>>> y = pd.Series([1.,np.nan,3.,4.], drange(-4,-1))
>>> assert eq(a(x,y), pd.Series([np.nan, 7], index))
```

Example 6 returning complicated stuff

```
>>> from pyg import *
>>> a = pd.DataFrame(np.random.normal(0,1,(100,10)), drange(-99))
>>> b = pd.DataFrame(np.random.normal(0,1,(100,10)), drange(-99))
```

```
>>> def f(a, b):
>>>     return (a*b, ts_sum(a), ts_sum(b))
```

```
>>> old = f(a,b)
>>> self = presync(f)
>>> args = (); kwargs = dict(a = a, b = b)
>>> new = self(*args, **kwargs)
>>> assert eq(new, old)
```

4.6.5 add/sub/mul/div/pow operators

`pyg.timeseries._index.`**`add_`**`(`*a*, *b*`)`
addition of a and b supporting presynching (inner join) of timeseries

`pyg.timeseries._index.`**`mul_`**`(`*a*, *b*`)`
multiplication of a and b supporting presynching (inner join) of timeseries

`pyg.timeseries._index.`**`div_`**`(`*a*, *b*`)`
division of a by b supporting presynching (inner join) of timeseries

`pyg.timeseries._index.`**`sub_`**`(`*a*, *b*`)`
subtraction of b from a supporting presynching (inner join) of timeseries

`pyg.timeseries._index.`**`pow_`**`(`*a*, *b*`)`
equivalent to a**b supporting presynching (inner join) of timeseries

CHAPTER FIVE

TUTORIALS

Below are some tutorials covering some aspects of pyg that may not be obvious. All the tutorials are active python notebooks available in the ../docs/lab/ directory.

CHAPTER

SIX

PYG.BASE.DICT

There are a few existing dict-extensions similar to Dict (a nice example is https://github.com/mewwts/addict) but Dict has a little more up its sleeve.

6.1 initialization

[1]:
```python
from pyg import *
Dict(a = 1, b = 2, c = 3)
```

[1]:
```
{'a': 1, 'b': 2, 'c': 3}
```

[2]:
```python
Dict(a = 1)(b = 2, c = 3)
```

[2]:
```
{'a': 1, 'b': 2, 'c': 3}
```

[3]:
```python
Dict(a = 1)(b = 2)(c = lambda a, b: a+b)
```

[3]:
```
{'a': 1, 'b': 2, 'c': 3}
```

[4]:
```python
Dict(a = 1) + dict(b = 2, c = 3)
```

[4]:
```
{'a': 1, 'b': 2, 'c': 3}
```

6.2 members access

[5]:
```python
d = Dict(a = 1, b = 2, c = 3)
```

[6]:
```python
d.a
```

[6]:
```
1
```

[7]:
```python
d['a', 'b']
```

[7]:
```
[1, 2]
```

[8]:
```python
d[['a', 'b']]
```

[8]:
```
{'a': 1, 'b': 2}
```

But the fun starts when Dict allows you to access **functions** of its keys:

[9]:
```
d[lambda a, b: a + b]
```

[9]:
```
3
```

It is important to note that be making d[‘a’, ‘b’] access both ‘a’ and ‘b’ keys, we abandon the right to have tuples as keys.

[10]:
```
d = Dict({('a','b') : 1})
import pytest
with pytest.raises(KeyError): # Dict will be trying to grab 'a' and 'b' separately
    d[('a','b')]
```

6.3 adding

[11]:
```
Dict(a = 1, b = 2) + dict(b = 3, c = 4) # like .update() but not in-place
```

[11]:
```
{'a': 1, 'b': 3, 'c': 4}
```

But addition is subtly different from update in the case of tree structure:

[12]:
```
tree = Dict(a = 1, b = Dict(c = 2, d = 3))
update = dict(x = 1, b = dict(c = 'new value for b.c but keep b.d', e = 4))
tree+update
```

[12]:
```
{'a': 1, 'b': {'c': 'new value for b.c but keep b.d', 'd': 3, 'e': 4}, 'x': 1}
```

Tree updating is actually important enough to have its own function that can operate on dict-trees

[13]:
```
tree = dict(a = 1, b = dict(c = 2, d = 3)) # I only use dicts
tree_update(tree, update) # but I can still update it like a tree
```

[13]:
```
{'a': 1, 'b': {'c': 'new value for b.c but keep b.d', 'd': 3, 'e': 4}, 'x': 1}
```

6.4 subtracting

You can subtract keys or list of keys

[23]:
```
Dict(a = 'remove me', b = 2, c = 3) - 'a'  # subtracting a key
```

[23]:
```
{'b': 2, 'c': 3}
```

[24]:
```
Dict(a = 'I am gone', b = 'and so am I', c = 3) - ['a', 'b'] # subtracting a␣
↪collection of keys
```

[24]:
```
{'c': 3}
```

[28]:
```
tree = Dict(a = 1, b = Dict(c = 'delete me', d = 'but keep me'), c = 3)
tree - ('b', 'c')  ## subtracting a branch in a tree using a tuple, possible because␣
↪we know ('b', 'c') is never a node
```

[28]:
```
{'a': 1, 'b': {'d': 'but keep me'}, 'c': 3}
```

6.5 modifying the keys: rename

```
[29]: Dict(a = 1, b = 2).rename('prefix_') # need to be done sufficient
```

```
[29]: {'prefix_a': 1, 'prefix_b': 2}
```

```
[30]: Dict(a = 1, b = 2).rename('_suffix')
```

```
[30]: {'a_suffix': 1, 'b_suffix': 2}
```

```
[31]: Dict(a = 1, b = 2).rename(upper)
```

```
[31]: {'A': 1, 'B': 2}
```

```
[33]: Dict(a = 1, b = 2, c = 3).rename(a = 'Abraham', b = 'Barbara')
```

```
[33]: {'Abraham': 1, 'Barbara': 2, 'c': 3}
```

6.6 modifying the values: do

```
[34]: Dict(a = 1, b = 2, c = 3).do(lambda x: x**2) # modify all values using function
```

```
[34]: {'a': 1, 'b': 4, 'c': 9}
```

```
[37]: Dict(a = 1, b = 2, c = 3).do([lambda x: x**2, lambda x: x-1]) # modify all values␣
↪using list of function
```

```
[37]: {'a': 0, 'b': 3, 'c': 8}
```

```
[40]: Dict(a = 1, b = 2, c = 3).do([lambda x: x**2, lambda x: x-1], 'a', 'b') # modify␣
↪selected keys using list of function
```

```
[40]: {'a': 0, 'b': 3, 'c': 3}
```

6.7 Dict can store a calculation flow

Being able to access function of members means we can think of a Dict as a container of variables. Consider this code:

```
[17]: def func(a, b):
          c = a + b
          d = b + c
          e = a/b + d/c
          f = (d+e)/c
          return f
      func(1,2)
```

```
[17]: 2.388888888888889
```

How can we keep track of our calculations and debug it easily? Consider rewriting this:

```
[18]: x = Dict(a = 1, b = 2)
      x = x(c = lambda a, b: a + b)
      x = x(d = lambda b, c: b + c)
```

(continues on next page)

(continued from previous page)

```
x = x(e = lambda a,b,c,d : a/b + d/c)
x = x(f = lambda c,d,e: (d+e)/c)
x
```

[18]:
```
{'a': 1,
 'b': 2,
 'c': 3,
 'd': 5,
 'e': 2.1666666666666667,
 'f': 2.388888888888889}
```

We have all the internals of the function exposed and we are able to separate calculation flow and data easily:

[19]:
```
calculation_pipeline = dict(c = lambda a, b: a + b,
                            d = lambda b, c: b + c,
                            e = lambda a,b,c,d : a/b + d/c,
                            f = lambda c,d,e: (d+e)/c)

initial_values = Dict(a = 1, b = 2)
```

[20]:
```
initial_values(**calculation_pipeline)
```

[20]:
```
{'a': 1,
 'b': 2,
 'c': 3,
 'd': 5,
 'e': 2.1666666666666667,
 'f': 2.388888888888889}
```

You can see in pyg.base.dictable tutorial how this is extended

CHAPTER

SEVEN

PYG.BASE.DICTABLE

dictable is a table, a collection of iterable records. It is also a dict with each key's value being a column. Why not use a pandas.DataFrame? pd.DataFrame leads a dual life:

- by day an index-based optimized numpy array supporting e.g. timeseries analytics etc.
- by night, a table with keys supporting filtering, aggregating, pivoting on keys as well as inner/outer joining on keys.

As a result, the pandas interface is somewhat cumbersome. Further, the DataFrame isn't really designed for containing more complicated objects within it. Conversely, dictable only tries to do the latter and is designed precisely for holding entire research process in one place. You can think of dictable as 'one level up' on a DataFrame: a dictable will handle thousands of data frames within it with ease. Indeed, dictable should be thought of as an 'organiser of research flow' rather than as an array of primitives. In general, each row will contain some keys indexing the experiment, while some keys will contain complicated objects: a pd.DataFrame, a timeseries, yield_curves, machine-learning experiments etc. The interface is succinct and extremely intuitive, allowing the user to concentrate on logic of the calculations rather than boilerplate.

7.1 Motivation: dictable as an organiser of research flow

We start with a simple motivating example. Here is a typical workflow:

[3]:
```
from pyg import *; import pandas as pd; import numpy as np
import yfinance as yf
```

[4]:
```
symbols = ['MSFT', 'WMT', 'TSLA', 'AAPL', 'BAD_SYMBOL', 'C']
history = [yf.download(symbol) for symbol in symbols]
prices = [h['Adj Close'] for h in history]
rtns = [p.diff() for p in prices]
vols = [r.ewm(30).std() for r in rtns]
zscores = [r/v for r,v in zip(rtns, vols)]
zavgs = [z.mean() for z in zscores]
```

```
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed

1 Failed download:
- BAD_SYMBOL: No data found, symbol may be delisted
[*********************100%***********************]  1 of 1 completed
```

```
[5]: zavgs
```

```
[5]: [0.06238896915574035,
 0.045634555332148996,
 0.0676156301672513,
 0.05318957566922761 4,
 nan,
 0.027297252361386543]
```

At this point we ask ourselves: Why do we have a **nan**? Which ticker was it, and when did it go wrong?

```
[6]: bad_symbols = [s for s, z in zip(symbols, zavgs) if np.isnan(z)]; bad_symbols
```

```
[6]: ['BAD_SYMBOL']
```

Great, how do we remove bad symbols from all our other variables?

```
[7]: vols = [v for s, v in zip(symbols, vols) if s not in bad_symbols]
```

Now we can calculate some stuff with rtns and vols perhaps?

```
[8]: ewmas = [r.ewm(n).mean()/v for r,v in zip(rtns, vols) for n in [10, 20, 30]]
```

Things went wrong and went wrong silently too:

- We forgot to remove bad data from rtns as well as from vols so our zip function is zipping the wrong stocks together
- It is nearly impossible to discover what item in the list belong to what n and what stock

If you ever dealt with real data, the mess described above must be familiar.

7.2 Same code, in dictable

```
[9]: from pyg import *
import yfinance as yf
s = dictable(symbol = ['MSFT', 'WMT', 'TSLA', 'AAPL', 'BAD_SYMBOL', 'C'])
s = s(history = lambda symbol: yf.download(symbol))
s = s(price = lambda history: history['Adj Close'])
s = s(rtn = lambda price: price.diff())
s = s(vol = lambda rtn: rtn.ewm(30).std())
s = s(zscore = lambda rtn, vol: rtn/vol)
s = s(zavg = lambda zscore: zscore.mean())
```

```
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed

1 Failed download:
- BAD_SYMBOL: No data found, symbol may be delisted
[*********************100%***********************]  1 of 1 completed
```

dictable **s** contains all our data.

- each row contains all the variables associated with a specific symbol
- each column corresponds to a variable

- adding a new variable is declarative and free of boiler-plate loop and zip

[10]:
```
s[['symbol', 'history', 'vol', 'zavg']]
```

[10]:
```
dictable[6 x 4]
symbol    |history                                               |vol               ␣
↪             |zavg
MSFT      |              Open        High         Low    |Date              ␣
↪             |0.06238896915574035
          |Date                                              |1986-03-13        NaN ␣
↪             |
          |1986-03-13    0.088542    0.101563    0.088542    |1986-03-14        NaN ␣
↪             |
          |1986-03-14    0.097222    0.102431    0.097222    |1986-03-17   0.000779 ␣
↪             |
          |1986-03-17    0.100694    0.103299    0.100694    |1986-03-18   0.001997 ␣
↪             |
WMT       |              Open        High         Low    |Date              ␣
↪             |0.045634555332148996
          |Date                                              |1972-08-25        NaN ␣
↪             |
          |1972-08-25    0.063477    0.064697    0.063477    |1972-08-28        NaN ␣
↪             |
          |1972-08-28    0.064453    0.064941    0.064209    |1972-08-29   0.000198 ␣
↪             |
          |1972-08-29    0.063965    0.063965    0.063477    |1972-08-30   0.000215 ␣
↪             |
TSLA      |              Open        High         Low    |Date              ␣
↪             |0.0676156301672513
          |Date                                              |2010-06-29         NaN␣
↪             |
          |2010-06-29    3.800000    5.000000    3.508000    |2010-06-30         NaN␣
↪             |
          |2010-06-30    5.158000    6.084000    4.660000    |2010-07-01    0.255972␣
↪             |
          |2010-07-01    5.000000    5.184000    4.054000    |2010-07-02    0.274164␣
↪             |
AAPL      |              Open        High         Low    |Date              ␣
↪             |0.053189575669227614
          |Date                                              |1980-12-12        NaN ␣
↪             |
          |1980-12-12    0.128348    0.128906    0.128348    |1980-12-15        NaN ␣
↪             |
          |1980-12-15    0.122210    0.122210    0.121652    |1980-12-16   0.001242 ␣
↪             |
          |1980-12-16    0.113281    0.113281    0.112723    |1980-12-17   0.004938 ␣
↪             |
BAD_SYMBOL|Empty DataFrame                                   |Series([], Name: Adj␣
↪Close, dtype: float64)|nan
          |Columns: [Open, High, Low, Close, Adj Close, Volum|               ␣
↪             |
          |Index: []                                         |               ␣
↪             |
C         |             Open       High        Low       C|Date              ␣
↪             |0.027297252361386543
          |Date                                              |1977-01-03        NaN ␣
↪             |
          |1977-01-03  16.133125  16.236876  16.133125  16.23|1977-01-04        NaN ␣
↪             |
```

(continues on next page)

(continued from previous page)

```
          |1977-01-04  16.236876  16.288750  16.184999  16.28|1977-01-05    0.053720 ␣
↪                    |
          |1977-01-05  16.288750  16.288750  16.133125  16.18|1977-01-06    0.043500 ␣
↪                    |
```

[11]:
```
s.zavg
```

[11]:
```
[0.06238896915574035,
 0.045634555332148996,
 0.0676156301672513,
 0.053189575669227614,
 nan,
 0.027297252361386543]
```

7.2.1 Oh, no, we have a bad symbol, how do we remove it?

[12]:
```
s = s.exc(zavg = np.nan); s.zavg
```

[12]:
```
[0.06238896915574035,
 0.045634555332148996,
 0.0676156301672513,
 0.053189575669227614,
 0.027297252361386543]
```

7.2.2 Now if we want to calculate something per symbol and window. . .

We want to create a new table, now keyed on two values: symbol and window n, so we create a bigger table using cross product:

[13]:
```
sn = s * dict(n = [10,20,30]) ## each row is now unique per symbol and window n
```

[14]:
```
sn = sn(ewma = lambda rtn, n, vol: rtn.ewm(n).mean()/vol)
```

And here is Citibank's three ewma. . .

[15]:
```
sn.inc(symbol = 'C')[['n', 'ewma']]
```

[15]:
```
dictable[3 x 2]
n  |ewma
10|Date
  |1977-01-03         NaN
  |1977-01-04         NaN
  |1977-01-05   -0.269415
  |1977-01-06   -0.636750
20|Date
  |1977-01-03         NaN
  |1977-01-04         NaN
  |1977-01-05   -0.252990
  |1977-01-06   -0.610388
30|Date
  |1977-01-03         NaN
  |1977-01-04         NaN
  |1977-01-05   -0.247336
  |1977-01-06   -0.601208
```

Here is a pivot table of the average of each ewma per symbol and window... Note that again, we can access functions of variables and not just the existing keys in the dictable

```
[16]: sn.pivot('symbol', 'n', lambda ewma: ewma.mean())
```

```
[16]: dictable[5 x 4]
symbol|10                    |20                    |30
AAPL  |[0.05186739363086048] |[0.048216725636102103]|[0.044663941035992756]
C     |[0.027514106874753336]|[0.026447710541888325]|[0.02513228496004374]
MSFT  |[0.061144190254947106]|[0.058806269926441355]|[0.056647557919386575]
TSLA  |[0.05809192519536144] |[0.0513719449570986]  |[0.0461212842741293]
WMT   |[0.04534668336293025] |[0.04406462686679985] |[0.0426962176790935]
```

7.3 dictable functionality

7.3.1 construction

dictable is quite flexible on constuctions.

```
[17]: d = dictable(a = [1,2,3,4], b = ['a', 'b', 'c', 'd']); d
```

```
[17]: dictable[4 x 2]
a|b
1|a
2|b
3|c
4|d
```

```
[18]: d = dictable(dict(a = [1,2,3,4], b = ['a', 'b', 'c', 'd']), symbol = ['MSFT', 'AAPL',
↪'APA', 'MMM'], exchange = 'NYSE'); d
```

```
[18]: dictable[4 x 4]
symbol|exchange|a|b
MSFT  |NYSE    |1|a
AAPL  |NYSE    |2|b
APA   |NYSE    |3|c
MMM   |NYSE    |4|d
```

```
[19]: df = pd.DataFrame(d) # can instantiate a DataFrame from a dictable with no code and␣
↪vice versa...
```

```
[20]: d = dictable(df); d
```

```
[20]: dictable[4 x 4]
symbol|exchange|a|b
MSFT  |NYSE    |1|a
AAPL  |NYSE    |2|b
APA   |NYSE    |3|c
MMM   |NYSE    |4|d
```

```
[21]: d = dictable([(1,3), (2,4), (3,5)], ['a', 'b']); d # construction from records as␣
↪tuples
```

```
[21]: dictable[3 x 2]
a|b
```

(continues on next page)

(continued from previous page)

```
1|3
2|4
3|5
```

```
[22]: d = dictable([dict(a = 1, b = 3), dict(a = 2, b = 4, d = 'new column'), dict(a = 3, b␣
      ↪= 5, c = 'also here')]); d # construction from records as dicts, mismatching on␣
      ↪keys is fine
```

```
[22]: dictable[3 x 4]
b|c        |a|d
3|None     |1|None
4|None     |2|new column
5|also here|3|None
```

```
[23]: d = dictable(read_csv('d:/dropbox/yoav/python/pyg/docs/constituents_csv.csv')); d =␣
      ↪d[:6]; d
```

```
[23]: dictable[6 x 3]
Symbol|Name                |Sector
MMM   |3M Company          |Industrials
AOS   |A.O. Smith Corp     |Industrials
ABT   |Abbott Laboratories|Health Care
ABBV  |AbbVie Inc.         |Health Care
ABMD  |ABIOMED Inc         |Health Care
ACN   |Accenture plc       |Information Technology
```

7.3.2 row access

```
[24]: d[0] #returns a record
```

```
[24]: {'Symbol': 'MMM', 'Name': '3M Company', 'Sector': 'Industrials'}
```

```
[25]: d[:2] ## subset rows using slice
```

```
[25]: dictable[2 x 3]
Symbol|Name           |Sector
MMM   |3M Company     |Industrials
AOS   |A.O. Smith Corp|Industrials
```

```
[26]: for row in d: # iteration is by row
          print(row)
```

```
{'Symbol': 'MMM', 'Name': '3M Company', 'Sector': 'Industrials'}
{'Symbol': 'AOS', 'Name': 'A.O. Smith Corp', 'Sector': 'Industrials'}
{'Symbol': 'ABT', 'Name': 'Abbott Laboratories', 'Sector': 'Health Care'}
{'Symbol': 'ABBV', 'Name': 'AbbVie Inc.', 'Sector': 'Health Care'}
{'Symbol': 'ABMD', 'Name': 'ABIOMED Inc', 'Sector': 'Health Care'}
{'Symbol': 'ACN', 'Name': 'Accenture plc', 'Sector': 'Information Technology'}
```

7.3.3 column access

[27]:
```
d.Name
```

[27]:
```
['3M Company',
 'A.O. Smith Corp',
 'Abbott Laboratories',
 'AbbVie Inc.',
 'ABIOMED Inc',
 'Accenture plc']
```

[28]:
```
d['Name']
```

[28]:
```
['3M Company',
 'A.O. Smith Corp',
 'Abbott Laboratories',
 'AbbVie Inc.',
 'ABIOMED Inc',
 'Accenture plc']
```

[29]:
```
d['Name', 'Sector']
```

[29]:
```
[('3M Company', 'Industrials'),
 ('A.O. Smith Corp', 'Industrials'),
 ('Abbott Laboratories', 'Health Care'),
 ('AbbVie Inc.', 'Health Care'),
 ('ABIOMED Inc', 'Health Care'),
 ('Accenture plc', 'Information Technology')]
```

[30]:
```
d[['Name', 'Sector']]
```

[30]:
```
dictable[6 x 2]
Name               |Sector
3M Company         |Industrials
A.O. Smith Corp    |Industrials
Abbott Laboratories|Health Care
AbbVie Inc.        |Health Care
ABIOMED Inc        |Health Care
Accenture plc      |Information Technology
```

7.3.4 d is a dict so supports the usual keys(), values() and items():

[31]:
```
for key, column in d.items():
    print(key, ':', column)
```

```
Symbol : ['MMM', 'AOS', 'ABT', 'ABBV', 'ABMD', 'ACN']
Name : ['3M Company', 'A.O. Smith Corp', 'Abbott Laboratories', 'AbbVie Inc.',
↪'ABIOMED Inc', 'Accenture plc']
Sector : ['Industrials', 'Industrials', 'Health Care', 'Health Care', 'Health Care',
↪'Information Technology']
```

access via **function** of variables is also supported

[32]:
```
d[lambda Symbol, Sector: '%s, %s'%(Symbol, Sector)]
```

[32]:
```
['MMM, Industrials',
 'AOS, Industrials',
```

(continues on next page)

(continued from previous page)

```
 'ABT, Health Care',
 'ABBV, Health Care',
 'ABMD, Health Care',
 'ACN, Information Technology']
```

7.3.5 column and row access are commutative

[33]:
```
assert d[0].Name == d.Name[0] == '3M Company'
assert d[0][lambda Symbol, Sector: '%s, %s'%(Symbol, Sector)] == d[lambda Symbol,␣
↪Sector: '%s, %s'%(Symbol, Sector)][0] == 'MMM, Industrials'
assert d[0]['Name'] == d['Name'][0]
assert d[:2]['Name', 'Sector'] == d['Name', 'Sector'][:2]
assert d[:2][['Name', 'Sector']] == d[['Name', 'Sector']][:2]
```

7.3.6 adding records

[34]:
```
d = dictable(name = ['alan', 'barbara', 'chris'], surname = ['abramson', 'brown',
↪'cohen'], age = [1,2,3])
```

[35]:
```
d + dict(name = 'david', surname = 'donaldson', age = 4) ## adding a single record
```

[35]:
```
dictable[4 x 3]
age|name   |surname
1  |alan   |abramson
2  |barbara|brown
3  |chris  |cohen
4  |david  |donaldson
```

[36]:
```
d + [dict(name = 'david', surname = 'donaldson', age = 4), dict(name = 'evan',␣
↪surname = 'emmerson', age = 5)]
```

[36]:
```
dictable[5 x 3]
age|name   |surname
1  |alan   |abramson
2  |barbara|brown
3  |chris  |cohen
4  |david  |donaldson
5  |evan   |emmerson
```

[37]:
```
d + dict(name = ['david', 'evan'], surname = ['donaldson', 'emmerson'], age = [4,5])
```

[37]:
```
dictable[5 x 3]
age|name   |surname
1  |alan   |abramson
2  |barbara|brown
3  |chris  |cohen
4  |david  |donaldson
5  |evan   |emmerson
```

[38]:
```
d + pd.DataFrame(dict(name = ['david', 'evan'], surname = ['donaldson', 'emmerson'],␣
↪age = [4,5]))
```

[38]:
```
dictable[5 x 3]
age|name   |surname
1  |alan   |abramson
2  |barbara|brown
3  |chris  |cohen
4  |david  |donaldson
5  |evan   |emmerson
```

7.3.7 adding/modifying columns

You can add a column or a constant by simply calling the dictable with the values:

[39]:
```
d(gender = ['m', 'f', 'm'])(school = 'St Paul')
```

[39]:
```
dictable[3 x 5]
name   |surname |age|gender|school
alan   |abramson|1  |m     |St Paul
barbara|brown   |2  |f     |St Paul
chris  |cohen   |3  |m     |St Paul
```

More interestingly, it can be a callable function using the other variables. . .

[40]:
```
d = d(initials = lambda name, surname: name[0] + surname[0]); d
```

[40]:
```
dictable[3 x 4]
name   |surname |age|initials
alan   |abramson|1  |aa
barbara|brown   |2  |bb
chris  |cohen   |3  |cc
```

Given d is a dict, a more traditional way of setting a new key is by simple assignment:

[41]:
```
d['initials'] = d[lambda name, surname: name[0] + surname[0]]; d
```

[41]:
```
dictable[3 x 4]
name   |surname |age|initials
alan   |abramson|1  |aa
barbara|brown   |2  |bb
chris  |cohen   |3  |cc
```

Or you can use the dict.update method:

[42]:
```
d.update(dict(gender = ['m', 'f', 'm'])); d
```

[42]:
```
dictable[3 x 5]
name   |surname |age|initials|gender
alan   |abramson|1  |aa      |m
barbara|brown   |2  |bb      |f
chris  |cohen   |3  |cc      |m
```

7.3.8 do

Sometime we want to apply the same function(s) to a collection of columns. For this, 'do' will do nicely:

```
[43]: d = d.do(upper, 'initials', 'gender').do(proper, 'name', 'surname'); d
```

```
[43]: dictable[3 x 5]
name   |surname |age|initials|gender
Alan   |Abramson|1  |AA      |M
Barbara|Brown   |2  |BB      |F
Chris  |Cohen   |3  |CC      |M
```

7.3.9 removing columns

```
[44]: d = d - 'initials'; d
```

```
[44]: dictable[3 x 4]
name   |surname |age|gender
Alan   |Abramson|1  |M
Barbara|Brown   |2  |F
Chris  |Cohen   |3  |M
```

7.3.10 removing rows

```
[45]: d.exc(name = 'Alan')
```

```
[45]: dictable[2 x 4]
age|gender|name   |surname
2  |F     |Barbara|Brown
3  |M     |Chris  |Cohen
```

```
[46]: d.inc(name = ['Alan', 'Chris'])
```

```
[46]: dictable[2 x 4]
age|gender|name |surname
1  |M     |Alan |Abramson
3  |M     |Chris|Cohen
```

```
[47]: d.inc(lambda age: age>1)
```

```
[47]: dictable[2 x 4]
age|gender|name   |surname
2  |F     |Barbara|Brown
3  |M     |Chris  |Cohen
```

```
[48]: d.exc(lambda gender: gender == 'M')
```

```
[48]: dictable[1 x 4]
name   |surname|age|gender
Barbara|Brown  |2  |F
```

```
[49]: d.exc(lambda name, surname: len(name)>len(surname))
```

```
[49]: dictable[2 x 4]
age|gender|name |surname
1   |M     |Alan |Abramson
3   |M     |Chris|Cohen
```

7.3.11 sort

```
[50]: d.sort('name', 'surname')
```

```
[50]: dictable[3 x 4]
name   |surname |age|gender
Alan   |Abramson|1  |M
Barbara|Brown   |2  |F
Chris  |Cohen   |3  |M
```

```
[51]: d.sort(lambda name: name[::-1]) # can sort on functions of variables too
```

```
[51]: dictable[3 x 4]
name   |surname |age|gender
Barbara|Brown   |2  |F
Alan   |Abramson|1  |M
Chris  |Cohen   |3  |M
```

7.3.12 listby(keys)

listby is like groupby except it returns a dictable with unique keys and the other columns are returned as a list. We find that MUCH more useful usually than groupby

```
[52]: grades = dictable(name = ['alan', 'barbara', 'chris'], grades = [30,90,80], subject =
↪'english', teacher = 'mr bennet') \
        + dictable(name = ['alan', 'david', 'esther'], grades = [40,50,70], subject =
↪'math', teacher = 'mrs ruler') \
        + dictable(name = ['barbara', 'chris', 'esther'], grades = [90,60,80], subject␣
↪= 'french', teacher = 'dr francois')
```

```
[53]: grades.listby('teacher')
```

```
[53]: dictable[3 x 4]
teacher    |grades      |name                         |subject
dr francois|[90, 60, 80]|['barbara', 'chris', 'esther']|['french', 'french', 'french']
mr bennet  |[30, 90, 80]|['alan', 'barbara', 'chris']  |['english', 'english',
↪'english']
mrs ruler  |[40, 50, 70]|['alan', 'david', 'esther']   |['math', 'math', 'math']
```

```
[54]: grades.listby('teacher')(avg_grade = lambda grades: np.mean(grades))
```

```
[54]: dictable[3 x 5]
teacher    |grades      |name                         |subject                       ␣
↪    |avg_grade
dr francois|[90, 60, 80]|['barbara', 'chris', 'esther']|['french', 'french', 'french
↪']   |76.66666666666667
mr bennet  |[30, 90, 80]|['alan', 'barbara', 'chris']  |['english', 'english',
↪'english']|66.66666666666667
mrs ruler  |[40, 50, 70]|['alan', 'david', 'esther']   |['math', 'math', 'math']      ␣
↪    |53.333333333333336
```

7.3.13 unlist

unlist undoes listby() assuming it is possible...

[55]:
```
grades.listby('teacher').unlist()
```

[55]:
```
dictable[9 x 4]
grades|name   |subject|teacher
90    |barbara|french |dr francois
60    |chris  |french |dr francois
80    |esther |french |dr francois
...9 rows...
40    |alan   |math   |mrs ruler
50    |david  |math   |mrs ruler
70    |esther |math   |mrs ruler
```

7.3.14 groupby(keys) and ungroup

This is similar to DatFrame groupby except that instead of a new object, a dictable is returned: The name of the grouped column is given by 'grp'. ungroup allows us to get back to original.

[56]:
```
classes = grades.groupby(['teacher', 'subject'], grp = 'class')
```

[57]:
```
classes[0]
```

[57]:
```
{'teacher': 'dr francois',
 'subject': 'french',
 'class': dictable[3 x 2]
 grades|name
 90    |barbara
 60    |chris
 80    |esther }
```

[58]:
```
classes.ungroup('class')
```

[58]:
```
dictable[9 x 4]
grades|name   |subject|teacher
90    |barbara|french |dr francois
60    |chris  |french |dr francois
80    |esther |french |dr francois
...9 rows...
40    |alan   |math   |mrs ruler
50    |david  |math   |mrs ruler
70    |esther |math   |mrs ruler
```

7.3.15 inner join

The multiplication operation is overloaded for the join method. By default, if two dictables share keys, the join is an inner join on the keys

[59]:
```
students = dictable(name = ['alan', 'barbara', 'chris', 'david', 'esthar', 'fabian'],␣
↪surname = ['abramsom', 'brown', 'cohen', 'drummond', 'ecklestone', 'fox'])
```

[60]:
```
print('shared keys:', grades.keys() & students.keys())
grades * students
```

```
shared keys: ['name']
```

[60]:
```
dictable[7 x 5]
name   |grades|subject|teacher    |surname
alan   |30    |english|mr bennet  |abramsom
alan   |40    |math   |mrs ruler  |abramsom
barbara|90    |english|mr bennet  |brown
...7 rows...
chris  |80    |english|mr bennet  |cohen
chris  |60    |french |dr francois|cohen
david  |50    |math   |mrs ruler  |drummond
```

Are there students with no surname? We can do a xor or use division which is overloaded for xor:

[61]:
```
grades / students
```

[61]:
```
dictable[2 x 4]
grades|name  |subject|teacher
70    |esther|math   |mrs ruler
80    |esther|french |dr francois
```

Are there students with no grades?

[62]:
```
students / grades
```

[62]:
```
dictable[2 x 2]
name  |surname
esthar|ecklestone
fabian|fox
```

[63]:
```
students = dictable(name = ['Alan', 'Barbara', 'Chris', 'David', 'Esther', 'Fabian'],␣
↪surname = ['abramsom', 'brown', 'cohen', 'drummond', 'ecklestone', 'fox'])
```

We fixed Esther's spelling but introduced capitalization, that is OK, we are allowed to inner join on functions of keys too.

[64]:
```
grades.join(students, 'name', lambda name: name.lower())
```

[64]:
```
dictable[9 x 5]
name   |grades|subject|teacher    |surname
alan   |30    |english|mr bennet  |abramsom
alan   |40    |math   |mrs ruler  |abramsom
barbara|90    |english|mr bennet  |brown
...9 rows...
david  |50    |math   |mrs ruler  |drummond
esther |70    |math   |mrs ruler  |ecklestone
esther |80    |french |dr francois|ecklestone
```

[65]:
```
students = dictable(first_name = ['alan', 'barbara', 'chris', 'david', 'esther',
↪'fabian'], surname = ['abramsom', 'brown', 'cohen', 'drummond', 'ecklestone', 'fox
↪'])
```

You can inner join on different column names and both columns will be populated:

[66]:
```
grades.join(pd.DataFrame(students), 'name', 'first_name')
```

[66]:
```
dictable[9 x 6]
name   |grades|subject|teacher    |first_name|surname
```

(continues on next page)

(continued from previous page)

```
alan   |30    |english|mr bennet  |alan      |abramsom
alan   |40    |math   |mrs ruler  |alan      |abramsom
barbara|90    |english|mr bennet  |barbara   |brown
...9 rows...
david  |50    |math   |mrs ruler  |david     |drummond
esther |70    |math   |mrs ruler  |esther    |ecklestone
esther |80    |french |dr francois|esther    |ecklestone
```

7.3.16 inner join (with other columns that match names)

By default, if columns are shared but are not in the join, they will be returned with a tuple containing both values

[67]:
```
x = dictable(key = ['a', 'b', 'c', 'c'], x = [1,2,3,4], y = [4,5,6,7])
y = dictable(key = ['b', 'b', 'c', 'a'], x = [1,2,3,4], z = [8,9,1,2])
x.join(y, 'key', 'key') ## ignore x column for joining
```

[67]:
```
dictable[5 x 4]
key|y|z|x
a  |4|2|(1, 4)
b  |5|8|(2, 1)
b  |5|9|(2, 2)
c  |6|1|(3, 3)
c  |7|1|(4, 3)
```

[68]:
```
x.join(y, 'key', 'key', mode = 'left') ## grab left value
```

[68]:
```
dictable[5 x 4]
key|y|z|x
a  |4|2|1
b  |5|8|2
b  |5|9|2
c  |6|1|3
c  |7|1|4
```

7.3.17 cross join

If no columns are shared, then a cross join is returned.

[69]:
```
x = dictable(x = [1,2,3,4])
y = dict(y = [1,2,3])
x * y
```

[69]:
```
dictable[12 x 2]
x|y
1|1
1|2
1|3
...12 rows...
4|1
4|2
4|3
```

[70]:
```
x.join(y, [], []) ## you can force a full outer join
```

[70]:
```
dictable[12 x 2]
x|y
1|1
1|2
1|3
...12 rows...
4|1
4|2
4|3
```

[71]:
```
x / y == x
```

[71]:
```
True
```

7.3.18 xor (versus left and right join)

We find left/right join actually not very useful. There is usually a genuine reason for records for which there is a match and for records for which there isn't. And the treatment of these is distinct, which means a left-join operation that joins the two outcomes together is positively harmful.

The xor operator is much more useful and you can use it to recreate left/right join if we really must. Here is an example

[80]:
```
students = dictable(name = ['alan', 'barbara', 'chris'], surname = ['abramsom', 'brown
↪', 'cohen',])
new_students = dictable(name = ['david', 'esther', 'fabian'], surname = ['drummond',
↪'ecklestone', 'fox'])

inner_join = grades * students ## grades with students
left_xor = grades / students  ## grades without sudents

# you can...
left_join = grades * students + grades / students ## grades for which no surname is␣
↪available will have None surname
left_join
```

[80]:
```
dictable[9 x 5]
grades|name   |teacher    |surname |subject
30    |alan   |mr bennet  |abramsom|english
40    |alan   |mrs ruler  |abramsom|math
90    |barbara|mr bennet  |brown   |english
...9 rows...
50    |david  |mrs ruler  |None    |math
70    |esther |mrs ruler  |None    |math
80    |esther |dr francois|None    |french
```

[73]:
```
# but really you want to do:
student_grades = grades * students
unmapped_grades = grades / students ## we treat this one separately...
new_student_grades =  unmapped_grades * new_students ## and grab surnames from the␣
↪new students table...
```

[74]:
```
assert len(unmapped_grades / new_student_grades) == 0, 'students must exist either in␣
↪the students table or in the new students table'
```

[75]:
```
all_grades = student_grades + new_student_grades; all_grades
```

```
[75]: dictable[9 x 5]
grades|name   |subject|surname   |teacher
30    |alan   |english|abramsom  |mr bennet
40    |alan   |math   |abramsom  |mrs ruler
90    |barbara|english|brown     |mr bennet
...9 rows...
50    |david  |math   |drummond  |mrs ruler
70    |esther |math   |ecklestone|mrs ruler
80    |esther |french |ecklestone|dr francois
```

7.3.19 pivot

```
[76]: x = dictable(x = [1,2,3,4])
y = dictable(y = [1,2,3,4])
xy = (x * y)
xy
```

```
[76]: dictable[16 x 2]
x|y
1|1
1|2
1|3
...16 rows...
4|2
4|3
4|4
```

```
[77]: xy.pivot('x', 'y', lambda x, y: x*y)
```

```
[77]: dictable[4 x 5]
x|1  |2  |3   |4
1|[1]|[2]|[3] |[4]
2|[2]|[4]|[6] |[8]
3|[3]|[6]|[9] |[12]
4|[4]|[8]|[12]|[16]
```

7.3.20 a few observations:

- as per usual, can provide a function for values in table (indeed columns y) and not just keys
- the output in the cells come back as a list. This is because sometimes there are more than one row with given x and y, and sometimes there are none:

```
[78]: (xy + xy).exc(lambda x,y: x+y == 5).pivot('x', 'y', lambda x, y: x*y)
```

```
[78]: dictable[4 x 5]
x|1     |2     |3       |4
1|[1, 1]|[2, 2]|[3, 3]  |None
2|[2, 2]|[4, 4]|None    |[8, 8]
3|[3, 3]|None  |[9, 9]  |[12, 12]
4|None  |[8, 8]|[12, 12]|[16, 16]
```

You can apply a sequence of aggregate functions:

[79]:
```
(xy + xy).exc(lambda x,y: x+y == 5).pivot('x', 'y', lambda x, y: x*y, lambda v:␣
↪len(v))
```

[79]:
```
dictable[4 x 5]
x|1   |2   |3   |4
1|2   |2   |2   |None
2|2   |2   |None|2
3|2   |None|2   |2
4|None|2   |2   |2
```

CHAPTER

EIGHT

PYG.MONGO

MongoDB has replaced our SQL databases as it is just too much fun to use. MongoDB does have its little quirks:

- The MongoDB 'query document' that replaces the SQL WHERE statements is very powerful but you need a PhD for even the simplest of queries.
- too many objects we use (specifically, numpy and pandas objects) cannot be pushed directly easily into Mongo.
- Mongo lacks the concept of a table with primary keys. Unstructured data is great but much of how we think of data is structured.

pyg.mongo addresses all three issues:

- **q** is a much easier way to generate Mongo queries. We are happy to acknowledge TinyDB https://tinydb.readthedocs.io/en/latest/usage.html#queries for the idea.
- **mongo_cursor** is a super-charged cursor and in particular, it handles encoding and decoding of objects seemlessly in a way that allows us to store all that we want in Mongo.
- **mongo_pk_cursor** manages a table with primary keys and full history audit. We are happy to acknowledge Arctic by the AHL Man team for the initial inspiration

8.1 q

The MongoDB interface for query of a collection (table) is via a creation of a query document https://docs.mongodb.com/manual/tutorial/query-documents/. This is rather complicated for the average use. For example, if you wanted to locate James Bond in the collection, you would need to compose q query document that looks like this:

[1]:
```
{"$and": [{"name": {"$eq": "James"}}, {"surname": {"$eq": "Bond"}}]}
```

[1]:
```
{'$and': [{'name': {'$eq': 'James'}}, {'surname': {'$eq': 'Bond'}}]}
```

It's doable, but not much fun writing. Luckily... within the continuum you can write this instead:

[2]:
```
from pyg import *; import re
q(name = 'James', surname = 'Bond')
```

[2]:
```
{"$and": [{"name": {"$eq": "James"}}, {"surname": {"$eq": "Bond"}}]}
```

[3]:
```
(q.name == 'James') & (q.surname == 'Bond')
```

[3]:
```
{"$and": [{"name": {"$eq": "James"}}, {"surname": {"$eq": "Bond"}}]}
```

How do we create in MongoDB a query document to find all the James who are not Bond?

[4]:
```
(q.surname!='Bond') & (q.name == 'James')
```

[4]:
```
{"$and": [{"name": {"$eq": "James"}}, {"surname": {"$ne": "Bond"}}]}
```

[5]:
```
~(q.surname=='Bond') & (q.name == 'James')
```

[5]:
```
{"$and": [{"$not": {"surname": {"$eq": "Bond"}}}, {"name": {"$eq": "James"}}]}
```

What about records with no surname?

[6]:
```
(q.name == 'James') - q.surname
```

[6]:
```
{"$and": [{"name": {"$eq": "James"}}, {"surname": {"$exists": false}}]}
```

[7]:
```
q(q.surname.not_exists, name = 'James')
```

[7]:
```
{"$and": [{"name": {"$eq": "James"}}, {"surname": {"$exists": false}}]}
```

And what about records with james rather than James?

[8]:
```
q(name = ['james', 'James'], surname = ['bond', 'Bond']) ## the result is long so it␣
↪is represented more nicely...
```

[8]:
```
$and:
    [{"name": {"$in": ["james", "James"]}}, {"surname": {"$in": ["bond", "Bond"]}}]
```

[9]:
```
q(name = re.compile('^[J|j]ames'), surname = re.compile('^[B|b]ond'))
```

[9]:
```
$and:
    [{"name": {"regex": "^[J|j]ames"}}, {"surname": {"regex": "^[B|b]ond"}}]
```

As you can see, q is callable and you can put expressions inside it, or you can use the q.key method.

If you have funny characters or spaces in your dict. . .

[10]:
```
q['funny$text with # weird £ characters'].exists
```

[10]:
```
{"funny$text with # weird £ characters": {"$exists": true}}
```

If your document is nested and there are subkeys, that is ok, you can use either:

[11]:
```
(q['key.subkey']>=100) | ((q.key.other.exists) & (q.some.other.stuff == [1,2]))
```

[11]:
```
$or:
    mdict
    $and:
        [{"key.other": {"$exists": true}}, {"some.other.stuff": {"$in": [1, 2]}}]
    M{"key.subkey": {"$gte": 100}}
```

q does not have the full power of the Mongo query document but it will get you to 95% of what you want. We end with a fun James Bond query. If we want to find the bond films with all actors who played James Bond after 1980. . .

[12]:
```
bonds = dictable(name = ['Daniel', 'Sean', 'Roger', 'Timothy'], surname = ['Craig',
↪'Connery', 'Moore', 'Dalton'])
bonds
```

[12]:
```
dictable[4 x 2]
name   |surname
Daniel |Craig
Sean   |Connery
```

(continues on next page)

(continued from previous page)

```
Roger  |Moore
Timothy|Dalton
```

```
[13]: q(list(bonds), q.release_date > dt(1980))
```

```
[13]: $and:
          mdict
          $or:
              M{"$and": [{"name": {"$eq": "Daniel"}}, {"surname": {"$eq": "Craig"}}]}
              M{"$and": [{"name": {"$eq": "Roger"}}, {"surname": {"$eq": "Moore"}}]}
              M{"$and": [{"name": {"$eq": "Sean"}}, {"surname": {"$eq": "Connery"}}]}
              M{"$and": [{"name": {"$eq": "Timothy"}}, {"surname": {"$eq": "Dalton"}}]}
          M{"release_date": {"$gt": datetime.datetime(1980, 1, 1, 0, 0)}}
```

8.2 mongo_cursor

The mongo cursor:

- enables saving seemlessly objects and data in MongoDB
- simplifies filtering
- simplifies projecting onto certain keys in document

8.2.1 general objects insertion into documents

pymongo.Collection supports insertion of documents into it:

```
[14]: from pyg import *; import pymongo as pym; import pytest
      c = pym.MongoClient()['test']['test']
      c.drop()                                    # drop all documents
      c.insert_one(dict(a = 1, b = 2))            # insert a document
```

```
[14]: <pymongo.results.InsertOneResult at 0x2d26bf25280>
```

```
[15]: assert c.count_documents({}) == 1  # in order to count documents, must apply the␣
      ↪empty query document {}
```

We can do similar stuff with a mongo_cursor:

```
[16]: t = mongo_table(table = 'test', db = 'test')
      t.drop()
      t.insert_one(dict(a = 1, b = 2))
```

```
2021-03-07 20:42:47,719 - pyg - INFO - INFO: deleting 1 documents based on M{}
```

```
[16]: {'a': 1, 'b': 2, '_id': ObjectId('60453ac70e096da27d7d20bf')}
```

```
[17]: assert len(t) == 1 #no need to specify the filter, mongo_cursor keeps track of the␣
      ↪current filter
```

Annoyingly, raw pymongo.Collection cannot encode for lots of existing objects.

[18]:
```python
ts = pd.Series([1.,2.], drange(2000,1))
a = np.arange(3)
f = np.float32(32.0)
with pytest.raises(Exception):
    c.insert_one(dict(a = a)) # cannot insert an array
with pytest.raises(Exception):
    c.insert_one(dict(f = f)) # cannot insert a numpy float, string or bool
with pytest.raises(Exception):
    c.insert_one(dict(ts = ts))  # cannot insert a pd.Series or DataFrame
```

Further, unless we define the encoding, new classes do not work either

[19]:
```python
class NewClass():
    def __init__(self, n):
        self.n = n
    def __eq__(self, other):
        return type(other) == type(self) and self.n == other.n
n = NewClass(1)
with pytest.raises(Exception):
    c.insert_one(dict(n = n))
```

Luckily, the mongo_cursor t can insert all these happily:

[20]:
```python
t.drop()
t.insert_one(Dict(a = a, f = f, ts = ts, n = n))
assert len(t) == 1
t[0] ## reading it back
```

```
2021-03-07 20:42:47,836 - pyg - INFO - INFO: deleting 1 documents based on M{}
```

[20]:
```
{'_id': ObjectId('60453ac70e096da27d7d20c4'),
 'a': array([0, 1, 2]),
 'f': 32.0,
 'ts': 2000-01-01    1.0
 2000-01-02    2.0
 dtype: float64,
 'n': <__main__.NewClass at 0x2d26c6439d0>}
```

8.2.2 document reading

What is nice is that when you read the document using the mongo_cursor, you get back the **object** you saved, not just the data. Is this magic? Not really… We read the doc directly from the Collection:

[21]:
```python
raw_doc = c.find_one({})
assert raw_doc['n'] == '{"py/object": "__main__.NewClass", "n": 1}'
assert encode(n) == '{"py/object": "__main__.NewClass", "n": 1}'
assert decode('{"py/object": "__main__.NewClass", "n": 1}') == n
assert t.writer == encode
assert t.reader == decode
```

- When writing, the mongo_cursor encodes the objects pre-saving it into Mongo, in this case as a simple dict
- When reading, it uses decode to convert what it reads back into the object
- This is done transparently though you can have full control via specifying writer/reader functions

This all works with the assumption that the person loading and the person saving share the library so objects can be instantiated on load. If construction method has changed and the object is not back-compatible, then user will receive the undecoded object and a warning message is logged.

8.2.3 document writing to files

MongoDB is great for manipulating/searching dict keys/values. The actual dataframes in each doc, we may want to save in a file system because:

- DataFrames are stored as bytes in MongoDB anyway, so they are not searchable
- Storing in files allows other non-python/non-MongoDB users easier access, allowing data to be detached from app
- MongoDB free version has limitations on size of document
- For data licensing issues, data must not sit on servers but needs to be stored on local computer

[22]:
```
t2 = mongo_table('test', 'test', writer = 'parquet')
t2.drop()
doc = dict(root = 'c:/temp', a = [a,a,a], ts = dict(one = ts, two = ts), f = f, n =␣
↪n)  ## can handle lists of arrays or dicts of stuff
t2.insert_one(doc)
encoded = c.find_one({})
print(tree_repr(encoded))
```

```
2021-03-07 20:42:47,907 - pyg - INFO - INFO: deleting 1 documents based on M{}
```

```
_id:
    60453ac90e096da27d7d20c6
root:
    c:/temp
a:
    {'_obj': '{"py/function": "numpy.load"}', 'file': 'c:/temp/a/0.npy'}
    {'_obj': '{"py/function": "numpy.load"}', 'file': 'c:/temp/a/1.npy'}
    {'_obj': '{"py/function": "numpy.load"}', 'file': 'c:/temp/a/2.npy'}
ts:
    one:
        _obj:
            {"py/function": "pyg.base._parquet.pd_read_parquet"}
        path:
            c:/temp/ts/one.parquet
    two:
        _obj:
            {"py/function": "pyg.base._parquet.pd_read_parquet"}
        path:
            c:/temp/ts/two.parquet
f:
    32.0
n:
    {"py/object": "__main__.NewClass", "n": 1}
```

You can see that starting at the root location, the document's numpy arrays and pandas have been saved to .npy and .parquet files

[23]:
```
print(tree_repr(decode(encoded)))
```

```
_id:
    60453ac90e096da27d7d20c6
root:
    c:/temp
```

(continues on next page)

(continued from previous page)

```
a:
    [array([0, 1, 2]), array([0, 1, 2]), array([0, 1, 2])]
ts:
    one:
        index
        2000-01-01    1.0
        2000-01-02    2.0
        dtype: float64
    two:
        index
        2000-01-01    1.0
        2000-01-02    2.0
        dtype: float64
f:
    32.0
n:
    <__main__.NewClass object at 0x000002D26CAFEC10>
```

[24]:
```
np.load('c:/temp/a/2.npy') ## can load data directly
```

[24]:
```
array([0, 1, 2])
```

[25]:
```
pd_read_parquet('c:/temp/ts/one.parquet')
```

[25]:
```
index
2000-01-01    1.0
2000-01-02    2.0
dtype: float64
```

8.2.4 document access

We start by pushing a 10x10 times table into t

[26]:
```
t.drop()
times_table = (dictable(a = range(10)) * dictable(b = range(10)))(c = lambda a, b:␣
↪a*b)
t.insert_many(times_table)
```

```
2021-03-07 20:42:49,637 - pyg - INFO - INFO: deleting 1 documents based on M{}
```

[26]:
```
<class 'pyg.mongo._cursor.mongo_cursor'> for Collection(Database(MongoClient(host=[
↪'localhost:27017'], document_class=dict, tz_aware=False, connect=True), 'test'),
↪'test')
M{} None
documents count: 100
dict_keys(['_id', 'a', 'b', 'c', '_obj'])
```

8.2.5 filters

We now examine how we drill down to the document(s) we want:

[27]:
```
assert len(t.inc(a = 1)) == 10
assert len(t.exc(a = 1)) == 90
assert isinstance(t.inc(a = 1), mongo_cursor) ## it is chain-able
assert len(t.find(q.a == 1).find(q.b == [1,2,3,4])) == 4
```

We can use the original collection too but not in a chain-like fashion:

[28]:
```
spec = q(a = 1, b = [1,2,3,4])
assert c.count_documents(spec) == 4
c.find(spec) # That is OK
with pytest.raises(AttributeError):  # not OK, cannot chain queries
    c.find(q(a=1)).find(q(b = [1,2,3,4]))
```

8.2.6 iteration

Just like a mongo.Cursor, c.find(spec), t is also iterable over the documents:

[29]:
```
sum([doc for doc in t.find(a = 1).find(b = [1,2,3,4])], dictable())
```

[29]:
```
dictable[4 x 4]
_id                     |a|b|c
60453ac90e096da27d7d20d2|1|1|1
60453ac90e096da27d7d20d3|1|2|2
60453ac90e096da27d7d20d4|1|3|3
60453ac90e096da27d7d20d5|1|4|4
```

[30]:
```
dictable(t.find(a = 1).find(b = [1,2,3,4])) ## or just put a cursor straight into a␣
↪table
```

[30]:
```
dictable[4 x 4]
_id                     |a|b|c
60453ac90e096da27d7d20d2|1|1|1
60453ac90e096da27d7d20d3|1|2|2
60453ac90e096da27d7d20d4|1|3|3
60453ac90e096da27d7d20d5|1|4|4
```

[31]:
```
t.find(a = 1).find(b = [1,2,3,4])[::] ## or simple slicing
```

[31]:
```
dictable[4 x 4]
_id                     |a|b|c
60453ac90e096da27d7d20d2|1|1|1
60453ac90e096da27d7d20d3|1|2|2
60453ac90e096da27d7d20d4|1|3|3
60453ac90e096da27d7d20d5|1|4|4
```

8.2.7 sorting

[32]:
```
t.sort('c', 'b')[::]
```

[32]:
```
dictable[100 x 4]
_id                     |a|b|c
60453ac90e096da27d7d20c7|0|0|0
60453ac90e096da27d7d20d1|1|0|0
60453ac90e096da27d7d20db|2|0|0
...100 rows...
60453ac90e096da27d7d2129|9|8|72
60453ac90e096da27d7d2120|8|9|72
60453ac90e096da27d7d212a|9|9|81
```

8.2.8 getitem of a specfic document

[33]:
```
t[dict(a = 7, b = 8)]
```

[33]:
```
{'_id': ObjectId('60453ac90e096da27d7d2115'), 'a': 7, 'b': 8, 'c': 56}
```

8.2.9 column access

[34]:
```
t.b
```

[34]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[35]:
```
assert t.b == t.distinct('b') == c.distinct('b')
```

In MongoDB the cursor can have a ‘projection’ onto specific columns. In mongo_cursor this is simplified:

[36]:
```
t[['a', 'b']].find(c = 12)[::]
```

[36]:
```
dictable[4 x 3]
_id                     |a|b
60453ac90e096da27d7d2105|6|2
60453ac90e096da27d7d20f2|4|3
60453ac90e096da27d7d20e9|3|4
60453ac90e096da27d7d20e1|2|6
```

8.2.10 add/remove columns

[37]:
```
del t['c']
t[::]
```

[37]:
```
dictable[100 x 3]
_id                     |a|b
60453ac90e096da27d7d20c7|0|0
60453ac90e096da27d7d20d1|1|0
60453ac90e096da27d7d20db|2|0
...100 rows...
60453ac90e096da27d7d2116|7|9
60453ac90e096da27d7d2120|8|9
60453ac90e096da27d7d212a|9|9
```

[38]:
```
t = t.set(c = 'not very useful but...')
t[::]
```

[38]:
```
dictable[100 x 4]
_id                     |a|b|c
60453ac90e096da27d7d20c7|0|0|not very useful but...
60453ac90e096da27d7d20d1|1|0|not very useful but...
60453ac90e096da27d7d20db|2|0|not very useful but...
...100 rows...
60453ac90e096da27d7d2116|7|9|not very useful but...
60453ac90e096da27d7d2120|8|9|not very useful but...
60453ac90e096da27d7d212a|9|9|not very useful but...
```

[39]:
```
t = t.set(c = lambda a, b: a * b) ### more useful
t[::]
```

[39]:
```
dictable[100 x 4]
_id                     |a|b|c
60453ac90e096da27d7d20c7|0|0|0
60453ac90e096da27d7d20d1|1|0|0
60453ac90e096da27d7d20db|2|0|0
...100 rows...
60453ac90e096da27d7d2129|9|8|72
60453ac90e096da27d7d2120|8|9|72
60453ac90e096da27d7d212a|9|9|81
```

8.2.11 add/remove records

[40]:
```
t.inc(c = 12).drop()
t
```

```
2021-03-07 20:42:51,028 - pyg - INFO - INFO: deleting 4 documents based on M{'c': {'
↪$eq': 12}}
```

[40]:
```
<class 'pyg.mongo._cursor.mongo_cursor'> for Collection(Database(MongoClient(host=[
↪'localhost:27017'], document_class=dict, tz_aware=False, connect=True), 'test'),
↪'test')
M{} None
documents count: 96
dict_keys(['_id', 'a', 'b', '_obj', 'c'])
```

[41]:
```
t = t + dict(a = 2, b = 6, c = 12)
t
```

[41]:
```
<class 'pyg.mongo._cursor.mongo_cursor'> for Collection(Database(MongoClient(host=[
↪'localhost:27017'], document_class=dict, tz_aware=False, connect=True), 'test'),
↪'test')
M{} None
documents count: 97
dict_keys(['_id', 'a', 'b', '_obj', 'c'])
```

[42]:
```
t = t.inc(c = 12).drop() + times_table.inc(c = 12) ## adding four records at once
t
```

```
2021-03-07 20:42:51,073 - pyg - INFO - INFO: deleting 1 documents based on M{'c': {'
↪$eq': 12}}
```

[42]:
```
<class 'pyg.mongo._cursor.mongo_cursor'> for Collection(Database(MongoClient(host=[
↪'localhost:27017'], document_class=dict, tz_aware=False, connect=True), 'test'),
↪'test')
M{'c': {'$eq': 12}} None
documents count: 4
dict_keys(['_id', 'a', 'b', 'c', '_obj'])
```

[43]:
```
t = t.inc(c = 12).drop().insert_many(times_table.inc(c = 12))
t[::]
```

```
2021-03-07 20:42:51,099 - pyg - INFO - INFO: deleting 4 documents based on M{'c': {'
↪$eq': 12}}
```

[43]:
```
dictable[4 x 4]
_id                     |a|b|c
60453acb0e096da27d7d2133|6|2|12
60453acb0e096da27d7d2132|4|3|12
60453acb0e096da27d7d2131|3|4|12
60453acb0e096da27d7d2130|2|6|12
```

[44]:
```
t = t.raw ## remove the filter c = 12
t
```

[44]:
```
<class 'pyg.mongo._cursor.mongo_cursor'> for Collection(Database(MongoClient(host=[
↪'localhost:27017'], document_class=dict, tz_aware=False, connect=True), 'test'),
↪'test')
M{} None
documents count: 100
dict_keys(['_id', 'a', 'b', '_obj', 'c'])
```

8.3 mongo_pk_table

mongo_pk_table is a mongo_cursor implementing a table with primary keys. Suppose we want to have a table of people:

[70]:
```
from pyg import *; import pymongo as pym; import pytest

t = mongo_table(table = 'test', db = 'test')
c = pym.MongoClient()['test']['test']
pk = mongo_table(table = 'test', db = 'test', pk = ['name', 'surname'])

t.drop()
d = dictable(name = ['alan', 'alan', 'barbara', 'chris'], surname = ['adams', 'jones',
↪ 'brown', 'jones'], age = [1,2,3,4])
pk.insert_many(d)
pk[::]
```

```
2021-03-07 21:04:34,232 - pyg - INFO - INFO: deleting 8 documents based on M{}
```

[70]:
```
dictable[4 x 5]
_id                     |_pk                |age|name   |surname
60453fe20e096da27d7d2150|['name', 'surname']|1  |alan   |adams
60453fe20e096da27d7d2151|['name', 'surname']|2  |alan   |jones
60453fe20e096da27d7d2152|['name', 'surname']|3  |barbara|brown
60453fe20e096da27d7d2153|['name', 'surname']|4  |chris  |jones
```

Now let us suppose a year has passed…

[71]:
```
pk.set(age = lambda age: age + 1)
pk[::]
```

[71]:
```
dictable[4 x 5]
_id                     |_pk                   |age|name   |surname
60453fe20e096da27d7d2150|['name', 'surname']|2  |alan   |adams
60453fe20e096da27d7d2151|['name', 'surname']|3  |alan   |jones
60453fe20e096da27d7d2152|['name', 'surname']|4  |barbara|brown
60453fe20e096da27d7d2153|['name', 'surname']|5  |chris  |jones
```

The pk-table actually maintains complete audit trail. Older records are not deleted, they just get '_deleted' parameter set for them.

[72]:
```
print(dictable(c))
```

```
_pk                 |name   |_obj                                |age|_deleted          ␣
↪         |_id                     |surname
['name', 'surname']|alan   |{"py/type": "pyg.base._dict.Dict"}|2  |None              ␣
↪         |60453fe20e096da27d7d2150|adams
['name', 'surname']|alan   |{"py/type": "pyg.base._dict.Dict"}|3  |None              ␣
↪         |60453fe20e096da27d7d2151|jones
['name', 'surname']|barbara|{"py/type": "pyg.base._dict.Dict"}|4  |None              ␣
↪         |60453fe20e096da27d7d2152|brown
['name', 'surname']|chris  |{"py/type": "pyg.base._dict.Dict"}|5  |None              ␣
↪         |60453fe20e096da27d7d2153|jones
['name', 'surname']|alan   |{"py/type": "pyg.base._dict.Dict"}|1  |2021-03-07 21:04:
↪34.284000|60453fe20e096da27d7d2154|adams
['name', 'surname']|alan   |{"py/type": "pyg.base._dict.Dict"}|2  |2021-03-07 21:04:
↪34.289000|60453fe20e096da27d7d2155|jones
['name', 'surname']|barbara|{"py/type": "pyg.base._dict.Dict"}|3  |2021-03-07 21:04:
↪34.293000|60453fe20e096da27d7d2156|brown
['name', 'surname']|chris  |{"py/type": "pyg.base._dict.Dict"}|4  |2021-03-07 21:04:
↪34.298000|60453fe20e096da27d7d2157|jones
```

You can see pk only looks at records where _deleted does not exist and _pk are set.

[73]:
```
pk
```

[73]:
```
<class 'pyg.mongo._pk_cursor.mongo_pk_cursor'> for␣
↪Collection(Database(MongoClient(host=['localhost:27017'], document_class=dict, tz_
↪aware=False, connect=True), 'test'), 'test')
M{'$and': [{"_deleted": {"$exists": false}}, {"_pk": {"$eq": ["name", "surname"]}}]}␣
↪None
documents count: 4
dict_keys(['_id', '_obj', '_pk', 'age', 'name', 'surname'])
```

There are obviourslу some small differences on how pk works but broadly, it is just like a normal mongo_cursor with an added filter to zoom onto the records that maintain the primary-key table:

- you cannot insert docs without primary keys all present:
- the drop() command does not actually delete the documents, they are simply 'marked' as deleted.
- to get from a mongo_pk_cursor to mongo_cursor, simply go pk.raw

[74]:
```
with pytest.raises(KeyError):
    pk.insert_one(dict(no_name_or_surname = 'James')) # cannot insert with no PK
```

[75]:
```
pk.drop()
len(pk)
```

[75]:
```
0
```

[76]:
```
t[::] ## the data is there, it is just marked as _deleted
```

[76]:
```
dictable[8 x 6]
_deleted                  |_id                     |_pk                 |age|name  ␣
↪|surname
2021-03-07 21:04:34.355000|60453fe20e096da27d7d2150|['name', 'surname']|2  |alan  ␣
↪|adams
2021-03-07 21:04:34.355000|60453fe20e096da27d7d2151|['name', 'surname']|3  |alan  ␣
↪|jones
2021-03-07 21:04:34.355000|60453fe20e096da27d7d2152|['name', 'surname']|4 ␣
↪|barbara|brown
...8 rows...
2021-03-07 21:04:34.289000|60453fe20e096da27d7d2155|['name', 'surname']|2  |alan  ␣
↪|jones
2021-03-07 21:04:34.293000|60453fe20e096da27d7d2156|['name', 'surname']|3 ␣
↪|barbara|brown
2021-03-07 21:04:34.298000|60453fe20e096da27d7d2157|['name', 'surname']|4  |chris ␣
↪|jones
```

8.4 mongo_reader and mongo_pk_reader

Because it is so easy to do stuff in MongoDB, we could easily cause damage to the date underlying. We therefore also introduced read-only versions for the mongo_cursor and pk_cursor:

[77]:
```
pkr = mongo_table(table = 'test', db = 'test', pk = ['name', 'surname'], mode = 'r')
pkr
```

[77]:
```
<class 'pyg.mongo._pk_reader.mongo_pk_reader'> for␣
↪Collection(Database(MongoClient(host=['localhost:27017'], document_class=dict, tz_
↪aware=False, connect=True), 'test'), 'test')
M{'$and': [{"_deleted": {"$exists": false}}, {"_pk": {"$eq": ["name", "surname"]}}]}␣
↪None
documents count: 0
```

[78]:
```
with pytest.raises(AttributeError):
    pkr.drop()
```

[80]:
```
r  = mongo_table(table = 'test', db = 'test', mode = 'r')
with pytest.raises(AttributeError):
    r.drop()

r[::]
```

[80]:
```
dictable[8 x 6]
_deleted                  |_id                     |_pk                 |age|name  ␣
↪|surname
2021-03-07 21:04:34.355000|60453fe20e096da27d7d2150|['name', 'surname']|2  |alan  ␣
↪|adams
2021-03-07 21:04:34.355000|60453fe20e096da27d7d2151|['name', 'surname']|3  |alan  ␣
↪|jones
2021-03-07 21:04:34.355000|60453fe20e096da27d7d2152|['name', 'surname']|4 ␣
↪|barbara|brown
```

(continues on next page)

(continued from previous page)

```
...8 rows...
2021-03-07 21:04:34.289000|60453fe20e096da27d7d2155|['name', 'surname']|2  |alan  ␣
↪|jones
2021-03-07 21:04:34.293000|60453fe20e096da27d7d2156|['name', 'surname']|3 ␣
↪|barbara|brown
2021-03-07 21:04:34.298000|60453fe20e096da27d7d2157|['name', 'surname']|4  |chris ␣
↪|jones
```

[ ]:

```

```

CHAPTER

NINE

PYG.BASE.CELL

cell is a dict that forms part of a calculation graph. Most usefully, db_cell is implemented to maintain persistency of the function output in MongoDB. Before we start, we will show a few examples of how a cell works. Then, we will build a toy example of trading stocks based on an exponentially weighted crossover.

- We will start by creating the system using pyg.base.dictable and pyg.timeseries.
- We then repeat the same code, this time modifying it slightly to save the data and calculation graph in MongoDB while running the calculation.
- We conclude by discussing the two approaches

9.1 Cell 101

[1]:
```
from pyg import *
a = cell(lambda x, y: x + y,  x = 1, y = 2)
b = cell(lambda x, y: x * y,  x = 2, y = a)
b
```

[1]:
```
cell
x:
    2
y:
    cell
    {'x': 1, 'y': 2, 'function': <function <lambda> at 0x000002A68A888940>}
function:
    <function <lambda> at 0x000002A68A888700>
```

[2]:
```
b.keys() ## b is a dict
```

[2]:
```
['x', 'y', 'function']
```

[3]:
```
b._args ## inputs
```

[3]:
```
['x', 'y']
```

[4]:
```
b._output ## where the output will go once we calculate it
```

[4]:
```
['data']
```

[5]:
```
assert b.run() ## b has not calculated yet... please run it
```

[6]:
```
b() # calculated object note b().data
```

[6]:
```
cell
x:
    2
y:
    cell
    x:
        1
    y:
        2
    function:
        <function <lambda> at 0x000002A68A888940>
    data:
        3
function:
    <function <lambda> at 0x000002A68A888700>
data:
    6
```

[7]:
```
assert not b().run() ## b has calculated now... no need to run it
```

[8]:
```
cell(lambda x, y: x ** y)(x = a, y = 2) # you can define the cell and then call it␣
↪with the values
```

[8]:
```
cell
function:
    <function <lambda> at 0x000002A68E0EF0D0>
x:
    cell
    x:
        1
    y:
        2
    function:
        <function <lambda> at 0x000002A68A888940>
    data:
        3
y:
    2
data:
    9
```

9.2 Workflow without saving to the database

[9]:
```
from pyg import *;
import yfinance as yf # see https://github.com/ranaroussi/yfinance
constituents = dictable(read_csv('d:/dropbox/yoav/python/pyg/docs/constituents_csv.csv
↪')).rename(lower) # downloaded from <https://datahub.io/core/s-and-p-500-companies
↪#resource-constituents>
constituents
```

[9]:
```
dictable[505 x 3]
symbol|name                  |sector
MMM   |3M Company            |Industrials
```

(continues on next page)

(continued from previous page)

```
AOS   |A.O. Smith Corp        |Industrials
ABT   |Abbott Laboratories    |Health Care
...505 rows...
ZBH   |Zimmer Biomet Holdings|Health Care
ZION  |Zions Bancorp          |Financials
ZTS   |Zoetis                 |Health Care
```

[10]:

```
stocks = constituents.inc(sector = 'Energy')
stocks
```

[10]:

```
dictable[26 x 3]
name              |sector|symbol
Apache Corporation|Energy|APA
Baker Hughes Co   |Energy|BKR
Cabot Oil & Gas   |Energy|COG
...26 rows...
TechnipFMC        |Energy|FTI
Valero Energy     |Energy|VLO
Williams Companies|Energy|WMB
```

[11]:

```
stocks = stocks(history = lambda symbol, sector, name: yf.download(tickers = symbol))
```

```
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed

1 Failed download:
- NBL: No data found, symbol may be delisted
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
```

[12]:

```
stocks = stocks.inc(lambda history: len(history)>0)
```

[13]:

```
stocks = stocks(adj = lambda history: getitem(value = history, key = 'Adj Close'))
```

[14]:
```
stocks = stocks(rtn = lambda adj: diff(a = adj))
```

[15]:
```
stocks = stocks(vol = lambda rtn: ewmstd(a = rtn, n = 30))
```

[33]:
```
_data = 'data'
def crossover_(a, fast, slow, vol, instate = None):
    state = Dict(fast = {}, slow = {}, vol = {}) if instate is None else instate
    fast_ewma_ = ewma_(a, fast, instate = state.fast)
    slow_ewma_ = ewma_(a, slow, instate = state.slow)
    raw_signal = fast_ewma_.data - slow_ewma_.data
    signal_rms = ewmrms_(raw_signal, vol, instate = state.vol)
    normalized = raw_signal/v2na(signal_rms.data)
    return Dict(data = normalized, state = Dict(fast = fast_ewma_.state, slow = slow_
↪ewma_.state, vol = signal_rms.state))

crossover_.output = ['data', 'state']

def crossover(a, fast, slow, vol, state = None):
    return crossover_(a, fast, slow, vol, instate = state)
```

9.2.1 some more functions to calculate the profits & loss as well as the signal/noise ratio

[17]:
```
def signal_pnl(signal, rtn, vol):
    return shift(signal) * (rtn/vol)

def information_ratio(pnl):
    return 16 * ts_mean(pnl) / ts_std(pnl)
```

[18]:
```
forecasts = stocks * dictable(fast = [2,4,8], slow = [6,12,24], forecast = ['fast',
↪'medium', 'slow'])
```

[19]:
```
forecasts = forecasts(signal = lambda rtn, fast, slow: crossover_(rtn, fast = fast,␣
↪slow = slow, vol = 30).data)
```

[20]:
```
forecasts = forecasts(pnl = lambda signal, rtn, vol: signal_pnl(signal = signal, rtn␣
↪= rtn, vol = vol))
```

[21]:
```
forecasts = forecasts(ir = lambda pnl: information_ratio(pnl = pnl))
```

[22]:
```
print(forecasts.pivot('symbol', 'forecast', 'ir', [last, f12]))
```

```
symbol|fast |medium|slow
APA   |0.13 |-0.03 |-0.10
BKR   |0.06 |-0.10 |-0.14
COG   |0.20 |0.12  |-0.02
COP   |-0.14|-0.17 |-0.18
CVX   |0.47 |0.30  |0.11
CXO   |0.01 |-0.14 |-0.34
DVN   |0.15 |0.15  |0.17
EOG   |0.16 |0.05  |-0.03
FANG  |0.00 |-0.06 |-0.18
FTI   |-0.18|-0.37 |-0.37
```

(continues on next page)

(continued from previous page)

```
HAL   |0.73 |0.47  |0.29
HES   |0.16 |0.03  |0.00
HFC   |0.86 |0.80  |0.71
KMI   |0.45 |0.13  |0.03
MPC   |0.21 |0.45  |0.72
MRO   |0.24 |0.16  |0.14
NOV   |0.10 |-0.04 |-0.04
OKE   |-0.22|-0.15 |-0.06
OXY   |-0.23|-0.29 |-0.22
PSX   |-0.02|0.11  |0.42
PXD   |0.22 |0.17  |0.23
SLB   |-0.08|-0.25 |-0.33
VLO   |0.30 |0.29  |0.41
WMB   |0.17 |-0.08 |-0.22
XOM   |-0.03|-0.28 |-0.43
```

9.3 Workflow while saving to MongoDB

9.3.1 Table creation

We create three tables dependending on the primary keys we will be using.

[23]:
```
idb = partial(mongo_table, db = 'demo', table = 'items', pk = 'item')
sdb = partial(mongo_table, db = 'demo', table = 'stock', pk = ['item', 'symbol'])
fdb = partial(mongo_table, db = 'demo', table = 'forecast', pk = ['item', 'symbol',
↪'forecast'])
```

[24]:
```
idb().insert_one(Dict(item = 'constituents', data = constituents))
```

[24]:
```
ObjectId('602a566073531499b3861809')
```

9.3.2 Any code differences?

Most of the code remains the same as above, except:

- We wrap it inside a periodic_cell so it is calculated daily
- We add reference to where we want to store it in MongoDB by specifying the db as well as the primary keys of that table
- To run the function, we need to call the cell. This: loads the cell from the database (if found), checking if it even needs running and if so, runs it.

[25]:
```
stocks = stocks(history = lambda symbol, sector, name: periodic_cell(yf.download,␣
↪tickers = symbol,       # these are the inputs for the function
                                          db = sdb, item = 'history', symbol =␣
↪symbol)())               # these define where the data goes to
```

```
2021-03-06 19:59:31,197 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'APA'))
```

```
[*********************100%***********************]  1 of 1 completed
```

```
2021-03-06 19:59:34,627 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'BKR'))
```

```
[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:35,356 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'COG'))
[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:36,122 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'CVX'))
[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:38,242 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'CXO'))
[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:38,754 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'COP'))
[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:39,444 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'DVN'))

[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:40,591 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'FANG'))
[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:40,978 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'EOG'))

[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:41,557 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'XOM'))
[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:44,056 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'HAL'))
[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:45,237 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'HES'))
[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:46,323 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'HFC'))
[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:47,026 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'KMI'))
[*********************100%***********************]  1 of 1 completed
2021-03-06 19:59:48,145 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'MRO'))
[*********************100%***********************]  1 of 1 completed
```

```
2021-03-06 19:59:50,428 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'MPC'))
```

```
[*********************100%***********************]  1 of 1 completed
```

```
2021-03-06 19:59:51,781 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'NOV'))
```

```
[*********************100%***********************]  1 of 1 completed
```

```
2021-03-06 19:59:52,541 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'OXY'))
```

```
[*********************100%***********************]  1 of 1 completed
```

```
2021-03-06 19:59:54,166 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'OKE'))
```

```
[*********************100%***********************]  1 of 1 completed
```

```
2021-03-06 19:59:55,333 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'PSX'))
```

```
[*********************100%***********************]  1 of 1 completed
```

```
2021-03-06 19:59:55,735 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'PXD'))
```

```

[*********************100%***********************]  1 of 1 completed
```

```
2021-03-06 19:59:56,563 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'SLB'))
```

```
[*********************100%***********************]  1 of 1 completed
```

```
2021-03-06 19:59:58,319 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'FTI'))
```

```
[*********************100%***********************]  1 of 1 completed
```

```
2021-03-06 19:59:58,844 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'VLO'))
```

```
[*********************100%***********************]  1 of 1 completed
```

```
2021-03-06 19:59:59,789 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('history', 'WMB'))
```

```
[*********************100%***********************]  1 of 1 completed
```

9.3.3 Accessing the data in MongoDB

The data is now in the database and can be accessed:

[26]:
```
get_data('stock', 'demo', symbol = 'MMM')
```

[26]:
```
                 Open      High       Low     Close  Adj Close  \
Date
1970-01-02   6.851563  6.890625  6.843750  6.851563   1.471232
1970-01-05   6.859375  6.898438  6.859375  6.890625   1.479620
1970-01-06   6.890625  6.960938  6.882813  6.960938   1.494717
1970-01-07   6.960938  7.015625  6.945313  7.000000   1.503106
1970-01-08   7.000000  7.109375  6.984375  7.093750   1.523237
```

(continues on next page)

(continued from previous page)

```
...                ...         ...         ...         ...         ...
2021-02-08  179.300003  180.869995  179.169998  180.759995  179.282608
2021-02-09  181.220001  181.899994  180.179993  180.940002  179.461151
2021-02-10  181.880005  182.380005  180.639999  181.080002  179.600006
2021-02-11  179.350006  179.880005  175.839996  177.210007  177.210007
2021-02-12  177.270004  178.839996  177.210007  178.699997  178.699997

             Volume
Date
1970-01-02    72000
1970-01-05   446400
1970-01-06   176000
1970-01-07   164800
1970-01-08   304000
...             ...
2021-02-08  2355100
2021-02-09  1942800
2021-02-10  1929000
2021-02-11  2187100
2021-02-12  1081500

[12895 rows x 6 columns]
```

[27]:

```
stocks = stocks.inc(lambda history: len(history.data)>0)
```

[28]:

```
stocks = stocks(adj = lambda history, symbol: periodic_cell(getitem, value = history,␣
↪key = 'Adj Close',
                                                                   db = sdb, symbol = symbol,
↪ item = 'adj')())
```

```
2021-03-06 20:00:02,042 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'APA'))
2021-03-06 20:00:03,128 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'BKR'))
2021-03-06 20:00:03,468 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'COG'))
2021-03-06 20:00:03,773 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'CVX'))
2021-03-06 20:00:04,094 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'CXO'))
2021-03-06 20:00:04,340 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'COP'))
2021-03-06 20:00:04,633 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'DVN'))
2021-03-06 20:00:04,936 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'FANG'))
2021-03-06 20:00:05,160 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'EOG'))
2021-03-06 20:00:05,518 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'XOM'))
2021-03-06 20:00:05,860 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'HAL'))
2021-03-06 20:00:06,272 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'HES'))
2021-03-06 20:00:06,881 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'HFC'))
```

(continues on next page)

(continued from previous page)

```
2021-03-06 20:00:07,289 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'KMI'))
2021-03-06 20:00:07,601 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'MRO'))
2021-03-06 20:00:07,905 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'MPC'))
2021-03-06 20:00:08,608 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'NOV'))
2021-03-06 20:00:09,233 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'OXY'))
2021-03-06 20:00:09,762 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'OKE'))
2021-03-06 20:00:10,238 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'PSX'))
2021-03-06 20:00:11,431 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'PXD'))
2021-03-06 20:00:13,090 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'SLB'))
2021-03-06 20:00:14,979 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'FTI'))
2021-03-06 20:00:15,767 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'VLO'))
2021-03-06 20:00:16,145 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('adj', 'WMB'))
```

[29]:
```
stocks = stocks(rtn = lambda adj, symbol: periodic_cell(diff, a = adj,
                                            db = sdb, symbol = symbol,␣
↪item = 'rtn')())
```

```
2021-03-06 20:00:16,480 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'APA'))
2021-03-06 20:00:16,930 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'BKR'))
2021-03-06 20:00:17,348 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'COG'))
2021-03-06 20:00:17,656 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'CVX'))
2021-03-06 20:00:18,515 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'CXO'))
2021-03-06 20:00:18,816 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'COP'))
2021-03-06 20:00:19,750 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'DVN'))
2021-03-06 20:00:20,084 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'FANG'))
2021-03-06 20:00:20,322 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'EOG'))
2021-03-06 20:00:20,813 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'XOM'))
2021-03-06 20:00:21,168 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'HAL'))
2021-03-06 20:00:21,636 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'HES'))
2021-03-06 20:00:22,050 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'HFC'))
2021-03-06 20:00:22,342 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'KMI'))
```

(continues on next page)

(continued from previous page)

```
2021-03-06 20:00:22,764 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'MRO'))
2021-03-06 20:00:23,127 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'MPC'))
2021-03-06 20:00:23,383 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'NOV'))
2021-03-06 20:00:23,681 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'OXY'))
2021-03-06 20:00:23,978 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'OKE'))
2021-03-06 20:00:24,327 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'PSX'))
2021-03-06 20:00:24,580 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'PXD'))
2021-03-06 20:00:24,850 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'SLB'))
2021-03-06 20:00:25,313 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'FTI'))
2021-03-06 20:00:25,611 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'VLO'))
2021-03-06 20:00:25,938 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('rtn', 'WMB'))
```

[30]:
```
stocks = stocks(vol = lambda rtn, symbol: periodic_cell(ewmstd, a = rtn, n =  30,
                                                 db = sdb, symbol = symbol,␣
↪item = 'vol')())
```

```
2021-03-06 20:00:26,300 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'APA'))
2021-03-06 20:00:26,595 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'BKR'))
2021-03-06 20:00:26,844 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'COG'))
2021-03-06 20:00:27,498 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'CVX'))
2021-03-06 20:00:28,895 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'CXO'))
2021-03-06 20:00:29,513 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'COP'))
2021-03-06 20:00:30,345 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'DVN'))
2021-03-06 20:00:31,002 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'FANG'))
2021-03-06 20:00:31,378 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'EOG'))
2021-03-06 20:00:31,711 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'XOM'))
2021-03-06 20:00:34,153 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'HAL'))
2021-03-06 20:00:35,635 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'HES'))
2021-03-06 20:00:36,165 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'HFC'))
2021-03-06 20:00:36,492 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'KMI'))
2021-03-06 20:00:37,163 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'MRO'))
```

(continues on next page)

(continued from previous page)

```
2021-03-06 20:00:37,823 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'MPC'))
2021-03-06 20:00:38,481 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'NOV'))
2021-03-06 20:00:39,119 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'OXY'))
2021-03-06 20:00:39,425 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'OKE'))
2021-03-06 20:00:39,705 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'PSX'))
2021-03-06 20:00:40,127 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'PXD'))
2021-03-06 20:00:40,378 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'SLB'))
2021-03-06 20:00:40,631 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'FTI'))
2021-03-06 20:00:41,056 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'VLO'))
2021-03-06 20:00:41,304 - pyg - INFO - ('localhost', 27017, 'demo', 'stock', ('item',
↪'symbol'), ('vol', 'WMB'))
```

9.3.4 Calculating the forecasts & saving them

[31]:
```
forecasts = stocks * dictable(fast = [2,4,8], slow = [6,12,24], forecast = ['fast',
↪'medium', 'slow'])
```

[34]:
```
forecasts = forecasts(signal = lambda rtn, fast, slow, symbol, forecast: periodic_
↪cell(crossover_, a = rtn, fast = fast, slow = slow, vol = 30,
                                            db = fdb, symbol = symbol, forecast =␣
↪forecast, item = 'signal')())
```

```
2021-03-06 20:04:15,290 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'APA'))
2021-03-06 20:04:52,869 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'APA'))
2021-03-06 20:04:53,186 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'APA'))
2021-03-06 20:04:53,434 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'BKR'))
2021-03-06 20:04:53,681 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'BKR'))
2021-03-06 20:04:53,913 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'BKR'))
2021-03-06 20:04:54,203 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'COG'))
2021-03-06 20:04:56,089 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'COG'))
2021-03-06 20:04:56,727 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'COG'))
2021-03-06 20:04:57,250 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'CVX'))
2021-03-06 20:04:58,763 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'CVX'))
2021-03-06 20:04:59,116 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'CVX'))
```

(continues on next page)

(continued from previous page)

```
2021-03-06 20:04:59,518 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'CXO'))
2021-03-06 20:05:00,010 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'CXO'))
2021-03-06 20:05:00,371 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'CXO'))
2021-03-06 20:05:00,771 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'COP'))
2021-03-06 20:05:01,088 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'COP'))
2021-03-06 20:05:01,374 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'COP'))
2021-03-06 20:05:01,673 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'DVN'))
2021-03-06 20:05:01,948 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'DVN'))
2021-03-06 20:05:02,256 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'DVN'))
2021-03-06 20:05:02,500 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'FANG'))
2021-03-06 20:05:02,708 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'FANG'))
2021-03-06 20:05:02,927 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'FANG'))
2021-03-06 20:05:03,134 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'EOG'))
2021-03-06 20:05:03,417 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'EOG'))
2021-03-06 20:05:03,671 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'EOG'))
2021-03-06 20:05:03,898 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'XOM'))
2021-03-06 20:05:04,224 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'XOM'))
2021-03-06 20:05:04,485 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'XOM'))
2021-03-06 20:05:04,701 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'HAL'))
2021-03-06 20:05:05,394 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'HAL'))
2021-03-06 20:05:05,652 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'HAL'))
2021-03-06 20:05:05,902 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'HES'))
2021-03-06 20:05:06,130 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'HES'))
2021-03-06 20:05:06,390 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'HES'))
2021-03-06 20:05:06,616 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'HFC'))
2021-03-06 20:05:06,883 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'HFC'))
2021-03-06 20:05:07,099 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'HFC'))
2021-03-06 20:05:07,321 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'KMI'))
2021-03-06 20:05:07,512 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'KMI'))
```

(continues on next page)

(continued from previous page)

```
2021-03-06 20:05:07,718 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'KMI'))
2021-03-06 20:05:07,941 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'MRO'))
2021-03-06 20:05:08,197 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'MRO'))
2021-03-06 20:05:08,458 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'MRO'))
2021-03-06 20:05:08,687 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'MPC'))
2021-03-06 20:05:08,905 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'MPC'))
2021-03-06 20:05:09,129 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'MPC'))
2021-03-06 20:05:09,348 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'NOV'))
2021-03-06 20:05:09,620 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'NOV'))
2021-03-06 20:05:09,845 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'NOV'))
2021-03-06 20:05:10,052 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'OXY'))
2021-03-06 20:05:10,383 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'OXY'))
2021-03-06 20:05:10,705 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'OXY'))
2021-03-06 20:05:10,958 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'OKE'))
2021-03-06 20:05:11,232 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'OKE'))
2021-03-06 20:05:11,475 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'OKE'))
2021-03-06 20:05:11,715 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'PSX'))
2021-03-06 20:05:11,997 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'PSX'))
2021-03-06 20:05:12,207 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'PSX'))
2021-03-06 20:05:12,414 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'PXD'))
2021-03-06 20:05:12,669 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'PXD'))
2021-03-06 20:05:12,901 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'PXD'))
2021-03-06 20:05:13,128 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'SLB'))
2021-03-06 20:05:13,751 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'SLB'))
2021-03-06 20:05:13,985 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'SLB'))
2021-03-06 20:05:14,230 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'FTI'))
2021-03-06 20:05:14,503 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'FTI'))
2021-03-06 20:05:14,775 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'FTI'))
2021-03-06 20:05:15,037 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'VLO'))
```

(continues on next page)

(continued from previous page)

```
2021-03-06 20:05:15,346 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'VLO'))
2021-03-06 20:05:15,578 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'VLO'))
2021-03-06 20:05:15,814 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'signal', 'WMB'))
2021-03-06 20:05:16,067 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'signal', 'WMB'))
2021-03-06 20:05:16,283 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'signal', 'WMB'))
```

[36]:
```
forecasts = forecasts(pnl = lambda signal, rtn, vol, symbol, forecast: periodic_
↪cell(signal_pnl, signal = signal, rtn = rtn, vol = vol,
                                                   db = fdb, symbol = symbol,␣
↪forecast = forecast, item = 'pnl')())
```

```
2021-03-06 20:07:41,567 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'APA'))
2021-03-06 20:09:01,609 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'APA'))
2021-03-06 20:09:02,368 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'APA'))
2021-03-06 20:09:03,885 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'BKR'))
2021-03-06 20:09:04,804 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'BKR'))
2021-03-06 20:09:05,484 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'BKR'))
2021-03-06 20:09:06,030 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'COG'))
2021-03-06 20:09:06,627 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'COG'))
2021-03-06 20:09:07,453 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'COG'))
2021-03-06 20:09:08,153 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'CVX'))
2021-03-06 20:09:09,125 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'CVX'))
2021-03-06 20:09:09,730 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'CVX'))
2021-03-06 20:09:10,582 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'CXO'))
2021-03-06 20:09:11,216 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'CXO'))
2021-03-06 20:09:11,904 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'CXO'))
2021-03-06 20:09:12,350 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'COP'))
2021-03-06 20:09:13,415 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'COP'))
2021-03-06 20:09:14,255 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'COP'))
2021-03-06 20:09:15,747 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'DVN'))
2021-03-06 20:09:16,408 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'DVN'))
2021-03-06 20:09:17,055 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'DVN'))
```

(continues on next page)

(continued from previous page)

```
2021-03-06 20:09:17,802 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'FANG'))
2021-03-06 20:09:18,207 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'FANG'))
2021-03-06 20:09:18,601 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'FANG'))
2021-03-06 20:09:20,074 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'EOG'))
2021-03-06 20:09:20,820 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'EOG'))
2021-03-06 20:09:21,605 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'EOG'))
2021-03-06 20:09:23,377 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'XOM'))
2021-03-06 20:09:25,407 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'XOM'))
2021-03-06 20:09:27,902 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'XOM'))
2021-03-06 20:09:29,626 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'HAL'))
2021-03-06 20:09:31,617 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'HAL'))
2021-03-06 20:09:32,908 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'HAL'))
2021-03-06 20:09:34,001 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'HES'))
2021-03-06 20:09:35,216 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'HES'))
2021-03-06 20:09:35,767 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'HES'))
2021-03-06 20:09:36,840 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'HFC'))
2021-03-06 20:09:38,454 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'HFC'))
2021-03-06 20:09:38,896 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'HFC'))
2021-03-06 20:09:39,445 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'KMI'))
2021-03-06 20:09:40,384 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'KMI'))
2021-03-06 20:09:40,983 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'KMI'))
2021-03-06 20:09:41,528 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'MRO'))
2021-03-06 20:09:43,164 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'MRO'))
2021-03-06 20:09:44,007 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'MRO'))
2021-03-06 20:09:44,933 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'MPC'))
2021-03-06 20:09:45,778 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'MPC'))
2021-03-06 20:09:46,376 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'MPC'))
2021-03-06 20:09:46,926 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'NOV'))
2021-03-06 20:09:47,990 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'NOV'))
```

(continues on next page)

(continued from previous page)

```
2021-03-06 20:09:48,697 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'NOV'))
2021-03-06 20:09:49,380 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'OXY'))
2021-03-06 20:09:50,123 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'OXY'))
2021-03-06 20:09:51,039 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'OXY'))
2021-03-06 20:09:52,698 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'OKE'))
2021-03-06 20:09:54,721 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'OKE'))
2021-03-06 20:09:55,441 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'OKE'))
2021-03-06 20:09:56,279 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'PSX'))
2021-03-06 20:09:56,974 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'PSX'))
2021-03-06 20:09:57,510 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'PSX'))
2021-03-06 20:09:57,980 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'PXD'))
2021-03-06 20:09:58,568 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'PXD'))
2021-03-06 20:09:59,271 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'PXD'))
2021-03-06 20:10:00,220 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'SLB'))
2021-03-06 20:10:02,318 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'SLB'))
2021-03-06 20:10:03,380 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'SLB'))
2021-03-06 20:10:04,161 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'FTI'))
2021-03-06 20:10:04,744 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'FTI'))
2021-03-06 20:10:05,535 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'FTI'))
2021-03-06 20:10:06,195 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'VLO'))
2021-03-06 20:10:07,851 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'VLO'))
2021-03-06 20:10:09,025 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'VLO'))
2021-03-06 20:10:10,054 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('fast', 'pnl', 'WMB'))
2021-03-06 20:10:11,128 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('medium', 'pnl', 'WMB'))
2021-03-06 20:10:11,790 - pyg - INFO - ('localhost', 27017, 'demo', 'forecast', (
↪'forecast', 'item', 'symbol'), ('slow', 'pnl', 'WMB'))
```

[37]:
```
forecasts = forecasts(ir = lambda pnl: information_ratio(pnl = pnl.data))
```

[38]:
```
print(forecasts.pivot('symbol', 'forecast', 'ir', [last, f12]))
```

```
symbol|fast |medium|slow
APA   |0.13 |-0.03 |-0.10
BKR   |0.06 |-0.10 |-0.14
COG   |0.20 |0.12  |-0.02
COP   |-0.14|-0.17 |-0.18
CVX   |0.47 |0.30  |0.11
CXO   |0.01 |-0.14 |-0.34
DVN   |0.15 |0.15  |0.17
EOG   |0.16 |0.05  |-0.03
FANG  |0.00 |-0.06 |-0.18
FTI   |-0.18|-0.37 |-0.37
HAL   |0.73 |0.47  |0.29
HES   |0.16 |0.03  |0.00
HFC   |0.86 |0.80  |0.71
KMI   |0.45 |0.13  |0.03
MPC   |0.21 |0.45  |0.72
MRO   |0.24 |0.16  |0.14
NOV   |0.10 |-0.04 |-0.04
OKE   |-0.22|-0.15 |-0.06
OXY   |-0.23|-0.29 |-0.22
PSX   |-0.02|0.11  |0.42
PXD   |0.22 |0.17  |0.23
SLB   |-0.08|-0.25 |-0.33
VLO   |0.30 |0.29  |0.41
WMB   |0.17 |-0.08 |-0.22
XOM   |-0.03|-0.28 |-0.43
```

9.3.5 Accessing & running the graph once the graph has been created

We can access the data or the cell:

```
[39]: get_cell('forecast', 'demo', symbol = 'APA', forecast = 'fast', item = 'signal')
```

```
[39]: periodic_cell
updated:
    2021-03-06 20:04:51.410000
_period:
    1b
db:
    functools.partial(<function mongo_table at 0x000002A68D03BCA0>, db='demo', table=
↪'forecast', pk=['item', 'symbol', 'forecast'])
_id:
    602a86fb4de6ffaf1c045c8f
_pk:
    ['forecast', 'item', 'symbol']
a:
    periodic_cell
    updated:
        None
    _period:
        1b
    db:
        functools.partial(<function mongo_table at 0x000002A68D03BCA0>, db='demo',␣
↪table='stock', pk=['item', 'symbol'])
    item:
        rtn
```

(continues on next page)

(continued from previous page)

```
    symbol:
        APA
    function:
        None
data:
    Date
    1979-05-15         NaN
    1979-05-16         NaN
    1979-05-17   -1.402762
    1979-05-18   -0.939334
    1979-05-21   -1.194304
                   ...
    2021-03-01   -1.018576
    2021-03-02   -0.831244
    2021-03-03   -0.187666
    2021-03-04    0.972348
    2021-03-05    2.674065
    Length: 10543, dtype: float64
fast:
    2
forecast:
    fast
instate:
    None
item:
    signal
slow:
    6
state:
    Dict
    fast:
        Dict
        state:
            {'t': nan, 't0': 0.9999999999999999, 't1': 0.46353936200681184}
        t:
            nan
        t0:
            0.9999999999999999
        t1:
            1.0413438705045697
    slow:
        Dict
        state:
            {'t': nan, 't0': 0.9999999999999997, 't1': 0.2672723621042594}
        t:
            nan
        t0:
            0.9999999999999997
        t1:
            0.5552029895351339
    vol:
        Dict
        state:
            {'t': nan, 't0': 0.9999999999999983, 't2': 0.02845240938173902}
        t:
            nan
        t0:
```

(continues on next page)

(continued from previous page)

```
            0.9999999999999983
        t2:
            0.033050680992027265
symbol:
    APA
vol:
    30
function:
    <function crossover_ at 0x000002A68E366C10>
```

And now that the graph has been created, you can actually trigger it just by loading. i.e. The code below will give you the fast signal for APA and will ensure it is up-to-date too:

[40]:
```
c = get_cell('forecast', 'demo', symbol = 'APA', forecast = 'fast', item = 'signal')
c = c.go()
print(c.data)
```

```
Date
1979-05-15         NaN
1979-05-16         NaN
1979-05-17   -1.402762
1979-05-18   -0.939334
1979-05-21   -1.194304
                ...
2021-03-01   -1.018576
2021-03-02   -0.831244
2021-03-03   -0.187666
2021-03-04    0.972348
2021-03-05    2.674065
Length: 10543, dtype: float64
```

9.4 Comparison of the two workflows

Saving to the database has negatives:

- does require some (but really not much) additional code to specify where each data item goes to
- slows down the calculation

Conversely,

- We get full persistency: We can access each part of the graph with full visibility on the inputs, the function used to calculate the result, the function output(s), the location of where the data is stored and the time it was last updated as well as the periodicity it is calculated.
- We get full audit, past calculations remain available to track if anything goes wrong
- Each node will manage its schedule, ensuring data is up-to-date
- We can run just the parts of the graph we are interested in (and can run in parallel)

9.5 Behind the scene: cell_func

Behind the scene of cell, there is machinary designed to make it work smoothly and transparently in most cases. However, sometimes the user may need to dig deeper. Here is an example for code that fails. . .

[50]:
```
from pyg import *
import pytest

def twox(x):
    return x*2
a = cell(a = 1)
c = cell(twox, x = a)

with pytest.raises(KeyError):
    c()
```

c tries to run the function. The function demands parameter x. When looking at the cells provided, cell 'a' does not contain anything like 'x' so the function fails.

[51]:
```
a = cell(data = 1)
cell(twox, x = a)()
```

[51]:
```
cell
x:
    cell
    {'data': 1, 'function': None}
function:
    <function twox at 0x000002A68A888430>
data:
    2
```

'data' key has a preferred status so although 'x' is not in the cell, we assume but default that 'data' parameter is the one the cell wants to present to the world. This is controlled by cell_output function:

[52]:
```
cell_output(a)
```

[52]:
```
['data']
```

[53]:
```
a = cell(data = 1, myoutput = 3, output = 'myoutput') ## you can decide your output␣
↪is different
cell_output(a), cell_item(a)
```

[53]:
```
(['myoutput'], 3)
```

[54]:
```
cell(twox, x = a)()
```

[54]:
```
cell
x:
    cell
    {'data': 1, 'myoutput': 3, 'function': None, 'output': 'myoutput'}
function:
    <function twox at 0x000002A68A888430>
data:
    6
```

That is good but what happens if the cell has MORE than one output or we want to direct the function to grab another key?

[55]:
```
a = cell(a = 1) ## this has failed...
cell(cell_func(twox, x = 'a'), x = a)() ## when you grab x, use 'a' as key
```

[55]:
```
cell
x:
    cell
    {'a': 1, 'function': None}
function:
    cell_func
    relabels:
        {'x': 'a'}
    unitemized:
        []
    uncalled:
        []
    function:
        <function twox at 0x000002A68A888430>
data:
    2
```

What if you need the cell itself rather than the items in it?

[56]:
```
def add_a_and_b(x):
    return x.a + x.b

x = cell(a = 1, b = 2)

cell(cell_func(add_a_and_b, unitemized = 'x'), x = x)()
```

[56]:
```
cell
x:
    cell
    {'a': 1, 'b': 2, 'function': None}
function:
    cell_func
    relabels:
        {}
    unitemized:
        ['x']
    uncalled:
        []
    function:
        <function add_a_and_b at 0x000002A6912A3160>
data:
    3
```

We can see that the cell x itself is presented to the function and x.a + x.b is calculated and data == 3

CHAPTER

TEN

PYG.BASE.JOIN

Only read this if you are a seasoned dictable user. In data science, we usually have data in multiple tables and we want to pull specific columns together for an analysis. We will first look at **join** function and then examine the **perdictable** decorator.

10.1 Join

10.1.1 Example: Using join function to transfer money to a bank

We begin by setting up a mini database:

[22]:
```
from pyg import *
customers = dictable(customer = ['alan', 'barbara', 'charles'], address = ['1 Abba␣
↪Avenue', '2 Beatles Lane', '3 Corrs Close'], bank = ['allied', 'barclays', 'chase'])
products = dictable(product = ['apple', 'banana', 'cherry'], price = [1,2,3],␣
↪supplier  = ['grove limited', 'go banabas', 'cherry pickers'])
customer_products = dictable(customer = ['alan', 'alan', 'charles', 'charles'],␣
↪product = ['apple', 'banana', 'cherry', 'apple'], amount = [1,2,3,4], purchase_date␣
↪= drange(-2,1))
banks = dictable(bank = ['allied', 'barclays'], account = [5556, 2461])

print('Customers\n', customers, '\n\nProducts\n', products, '\n\nCustomer_products\n',
↪ customer_products, '\n\nBanks\n', banks)
```

```
Customers
 customer|address       |bank
alan    |1 Abba Avenue |allied
barbara |2 Beatles Lane|barclays
charles |3 Corrs Close |chase

Products
 product|price|supplier
apple  |1    |grove limited
banana |2    |go banabas
cherry |3    |cherry pickers

Customer_products
 customer|product|amount|purchase_date
alan    |apple  |1     |2021-02-23 00:00:00
alan    |banana |2     |2021-02-24 00:00:00
charles |cherry |3     |2021-02-25 00:00:00
charles |apple  |4     |2021-02-26 00:00:00
```

(continues on next page)

(continued from previous page)

```
Banks
 bank    |account
allied  |5556
barclays|2461
```

10.1.2 Simple join: inner join between tables

Suppose we want to know how much money is to be transferred from each bank. - We only care about the fields 'bank', 'amount' and 'price' - each field is pulled from different tables, - need to specify customer & product as the keys we will join on:

```
[23]: join(dict(bank = customers, amount = customer_products, price = products), on = [
↪'customer',  'product'])
```

```
[23]: dictable[4 x 5]
product|customer|bank  |amount|price
apple  |alan    |allied|1     |1
banana |alan    |allied|2     |2
apple  |charles |chase |4     |1
cherry |charles |chase |3     |3
```

10.1.3 Defaults for fields we want to left-join on…

The function we need to run to transfer money looks like this, so actually, we would like to have account details too.

```
[24]: def transfer_money(bank, amount, price, account = 'default'):
          ## if account == 'default' transfer money slowly, else transfer quickly
          ## return
          pass
```

We can grab the account details from the 'banks' table:

```
[25]: join(dict(bank = customers, amount = customer_products, price = products, account =␣
↪banks), on = ['customer',  'product', 'bank'])
```

```
[25]: dictable[2 x 6]
bank  |product|customer|amount|price|account
allied|apple  |alan    |1     |1    |5556
allied|banana |alan    |2     |2    |5556
```

but we just **lost** Chase transactions as we dont have its account details. However, money is transfered perfectly (albeit slowly) even without account id. So instead….

```
[26]: join(dict(bank = customers, amount = customer_products, price = products, account =␣
↪banks),
          on = ['customer',  'product', 'bank'],
          defaults = dict(account = 'default'))
```

```
[26]: dictable[4 x 6]
account|amount|bank  |customer|price|product
5556   |1     |allied|alan    |1    |apple
5556   |2     |allied|alan    |2    |banana
default|4     |chase |charles |1    |apple
default|3     |chase |charles |3    |cherry
```

10.1.4 Renaming & calculating fields

We also want to ensure we don't transfer money that we already transferred... so we need to grab an expiry column based on purchase_date in customer_product table

```
[27]: join(dict(bank = customers, amount = customer_products, price = products, account =␣
↪banks, expiry = customer_products),
     on = ['customer',  'product', 'bank'],
     renames = dict(expiry = lambda purchase_date: dt(purchase_date, '1b')),   ## it␣
↪takes 1 business day to transfer money
     defaults = dict(account = 'default'))
```

```
[27]: dictable[4 x 7]
account|amount|bank  |customer|expiry             |price|product
5556   |1     |allied|alan    |2021-02-24 00:00:00|1    |apple
5556   |2     |allied|alan    |2021-02-25 00:00:00|2    |banana
default|4     |chase |charles |2021-03-01 00:00:00|1    |apple
default|3     |chase |charles |2021-02-26 00:00:00|3    |cherry
```

10.2 Perdictable

perdictable takes the same operation one steps further and actually runs the function. We also use the function signature to determine the defaults parameter. Here is another example: ### Example: Oil prices In Finance, there are contracts called Futures, each Future contract has an expiry. E.g. Futures contracts for Oil are contracts agreeing the delivery of oil to a particular place in a particular month. Once that month is gone, that contract is no longer traded and the oil needs to be delivered.

```
[28]: from pyg import *
oil = dictable(y = dt().year-1, m = range(3, 13, 3)) + dictable(y = dt().year, m =␣
↪range(3, 13, 3))
oil = oil(ticker = lambda y, m: 'OIL_%i_%s'%(y, m if m>9 else '0%i'%m))
oil
```

```
[28]: dictable[8 x 3]
m |y   |ticker
3 |2020|OIL_2020_03
6 |2020|OIL_2020_06
9 |2020|OIL_2020_09
...8 rows...
6 |2021|OIL_2021_06
9 |2021|OIL_2021_09
12|2021|OIL_2021_12
```

y,m and ticker will form our primary keys

```
[29]: pk = ['y', 'm', 'ticker']
expiry = perdictable(lambda y, m: dt(y,m+1,1), on = pk)(y = oil, m = oil)
expiry
```

```
[29]: dictable[8 x 4]
y   |m |ticker     |data
2020|3 |OIL_2020_03|2020-04-01 00:00:00
2020|6 |OIL_2020_06|2020-07-01 00:00:00
2020|9 |OIL_2020_09|2020-10-01 00:00:00
...8 rows...
2021|6 |OIL_2021_06|2021-07-01 00:00:00
```

(continues on next page)

(continued from previous page)

```
2021|9 |OIL_2021_09|2021-10-01 00:00:00
2021|12|OIL_2021_12|2022-01-01 00:00:00
```

[30]:
```
def fake_ts(ticker, expiry):
    return 500 + pd.Series(np.random.normal(0,1,100), drange(dt_bump(expiry,-99),␣
↪expiry)).cumsum()
```

To add a price for each of the futures, we first wrap fake_ts and then run it:

[31]:
```
price = perdictable(fake_ts, on = pk)(ticker = oil, expiry = expiry)
price
```

[31]:
```
dictable[8 x 4]
y    |m |ticker     |data
2020|3 |OIL_2020_03|2019-12-24    499.000139
    |  |           |2019-12-25    500.904180
    |  |           |2019-12-26    501.792007
    |  |           |2019-12-27    502.410313
    |  |           |2019-12-28    502.843697
2020|6 |OIL_2020_06|2020-03-24    500.575052
    |  |           |2020-03-25    499.504860
    |  |           |2020-03-26    500.558506
    |  |           |2020-03-27    500.599754
    |  |           |2020-03-28    500.704313
2020|9 |OIL_2020_09|2020-06-24    500.333677
    |  |           |2020-06-25    500.974220
    |  |           |2020-06-26    499.882500
    |  |           |2020-06-27    500.342359
    |  |           |2020-06-28    501.423622
...8 rows...
2021|6 |OIL_2021_06|2021-03-24    501.437903
    |  |           |2021-03-25    500.808820
    |  |           |2021-03-26    499.478861
    |  |           |2021-03-27    499.203311
    |  |           |2021-03-28    498.270609
2021|9 |OIL_2021_09|2021-06-24    499.950777
    |  |           |2021-06-25    500.458993
    |  |           |2021-06-26    498.564582
    |  |           |2021-06-27    497.988147
    |  |           |2021-06-28    498.193692
2021|12|OIL_2021_12|2021-09-24    500.320503
    |  |           |2021-09-25    499.915132
    |  |           |2021-09-26    498.200049
    |  |           |2021-09-27    497.805575
    |  |           |2021-09-28    497.879972
```

We have wrapped a function so that we get a price for **each** of these contracts. This allows us to move from operating on single timeseries, to run it on multiple rows from multiple tables

[32]:
```
rtn = perdictable(diff, on = pk)(a = price, expiry = expiry)
yesterday_price = perdictable(shift, on = pk)(a = price, expiry = expiry)
percentage_return = perdictable(div_, on = pk)(a = rtn, b = yesterday_price, expiry =␣
↪expiry)
percentage_return
```

[32]:
```
dictable[8 x 4]
y    |m |ticker     |data
```

(continues on next page)

(continued from previous page)

```
2020|3 |OIL_2020_03|2019-12-24        NaN
    |  |           |2019-12-25   0.003816
    |  |           |2019-12-26   0.001772
    |  |           |2019-12-27   0.001232
    |  |           |2019-12-28   0.000863
2020|6 |OIL_2020_06|2020-03-24        NaN
    |  |           |2020-03-25  -0.002138
    |  |           |2020-03-26   0.002109
    |  |           |2020-03-27   0.000082
    |  |           |2020-03-28   0.000209
2020|9 |OIL_2020_09|2020-06-24        NaN
    |  |           |2020-06-25   0.001280
    |  |           |2020-06-26  -0.002179
    |  |           |2020-06-27   0.000920
    |  |           |2020-06-28   0.002161
...8 rows...
2021|6 |OIL_2021_06|2021-03-24        NaN
    |  |           |2021-03-25  -0.001255
    |  |           |2021-03-26  -0.002656
    |  |           |2021-03-27  -0.000552
    |  |           |2021-03-28  -0.001868
2021|9 |OIL_2021_09|2021-06-24        NaN
    |  |           |2021-06-25   0.001017
    |  |           |2021-06-26  -0.003785
    |  |           |2021-06-27  -0.001156
    |  |           |2021-06-28   0.000413
2021|12|OIL_2021_12|2021-09-24        NaN
    |  |           |2021-09-25  -0.000810
    |  |           |2021-09-26  -0.003431
    |  |           |2021-09-27  -0.000792
    |  |           |2021-09-28   0.000149
```

10.2.1 perdictable and caching

This is nice but (a) what have we gained? and (b) why do we keep using expiry as a variable? The answer is to do with caching actually. If we rerun prices, we should get brand new data, since fake_ts just generates random prices… perdictable identifies rows that have been run and are now 'expired' It uses provided old data and does not recalculate. If either expiry or old values are not provided then it calculates everything.

[37]:
```
new_price = perdictable(fake_ts, on = pk)(ticker = oil, data = price, expiry = expiry)
(new_price.relabel(data = 'new') * price.relabel(data = 'old')).sort('y', 'm')
```

[37]:
```
dictable[8 x 5]
y   |m |ticker     |new                    |old
2020|3 |OIL_2020_03|2019-12-24   499.000139|2019-12-24   499.000139
    |  |           |2019-12-25   500.904180|2019-12-25   500.904180
    |  |           |2019-12-26   501.792007|2019-12-26   501.792007
    |  |           |2019-12-27   502.410313|2019-12-27   502.410313
    |  |           |2019-12-28   502.843697|2019-12-28   502.843697
2020|6 |OIL_2020_06|2020-03-24   500.575052|2020-03-24   500.575052
    |  |           |2020-03-25   499.504860|2020-03-25   499.504860
    |  |           |2020-03-26   500.558506|2020-03-26   500.558506
    |  |           |2020-03-27   500.599754|2020-03-27   500.599754
    |  |           |2020-03-28   500.704313|2020-03-28   500.704313
2020|9 |OIL_2020_09|2020-06-24   500.333677|2020-06-24   500.333677
```

(continues on next page)

(continued from previous page)

```
     |  |            |2020-06-25    500.974220|2020-06-25    500.974220
     |  |            |2020-06-26    499.882500|2020-06-26    499.882500
     |  |            |2020-06-27    500.342359|2020-06-27    500.342359
     |  |            |2020-06-28    501.423622|2020-06-28    501.423622
...8 rows...
2021|6 |OIL_2021_06|2021-03-24    500.429724|2021-03-24    501.437903
     |  |            |2021-03-25    501.537890|2021-03-25    500.808820
     |  |            |2021-03-26    501.167511|2021-03-26    499.478861
     |  |            |2021-03-27    502.611689|2021-03-27    499.203311
     |  |            |2021-03-28    501.820261|2021-03-28    498.270609
2021|9 |OIL_2021_09|2021-06-24    499.911914|2021-06-24    499.950777
     |  |            |2021-06-25    497.451472|2021-06-25    500.458993
     |  |            |2021-06-26    498.190816|2021-06-26    498.564582
     |  |            |2021-06-27    498.015362|2021-06-27    497.988147
     |  |            |2021-06-28    497.224958|2021-06-28    498.193692
2021|12|OIL_2021_12|2021-09-24    498.129511|2021-09-24    500.320503
     |  |            |2021-09-25    498.739546|2021-09-25    499.915132
     |  |            |2021-09-26    499.321094|2021-09-26    498.200049
     |  |            |2021-09-27    498.491587|2021-09-27    497.805575
     |  |            |2021-09-28    497.057529|2021-09-28    497.879972
```

10.2.2 perdictable with the cell framework

We can run the function and use a cell to store the output. . .

[40]:
```
c = cell(perdictable(fake_ts, on = pk), ticker = oil, expiry = expiry)()
c.data
```

[40]:
```
dictable[8 x 4]
y    |m |ticker      |data
2020|3 |OIL_2020_03|2019-12-24    501.331417
     |  |            |2019-12-25    500.332873
     |  |            |2019-12-26    500.160526
     |  |            |2019-12-27    496.688779
     |  |            |2019-12-28    497.774215
2020|6 |OIL_2020_06|2020-03-24    500.899756
     |  |            |2020-03-25    500.830490
     |  |            |2020-03-26    501.829020
     |  |            |2020-03-27    501.875464
     |  |            |2020-03-28    503.241949
2020|9 |OIL_2020_09|2020-06-24    500.395880
     |  |            |2020-06-25    500.311780
     |  |            |2020-06-26    499.817331
     |  |            |2020-06-27    499.780468
     |  |            |2020-06-28    497.550235
...8 rows...
2021|6 |OIL_2021_06|2021-03-24    501.291426
     |  |            |2021-03-25    499.592175
     |  |            |2021-03-26    499.104934
     |  |            |2021-03-27    497.698320
     |  |            |2021-03-28    497.868177
2021|9 |OIL_2021_09|2021-06-24    499.978264
     |  |            |2021-06-25    500.784927
     |  |            |2021-06-26    501.212177
     |  |            |2021-06-27    501.852472
```

(continues on next page)

(continued from previous page)

```
    |  |           |2021-06-28    502.035097
2021|12|OIL_2021_12|2021-09-24    500.282482
    |  |           |2021-09-25    501.077309
    |  |           |2021-09-26    501.005312
    |  |           |2021-09-27    501.173168
    |  |           |2021-09-28    502.098126
```

[43]:
```
recalculated_cell = c.go(1) ## force a recalculation
recalculated_cell.data
```

[43]:
```
dictable[8 x 4]
y   |m |ticker     |data
2020|3 |OIL_2020_03|2019-12-24    501.331417
    |  |           |2019-12-25    500.332873
    |  |           |2019-12-26    500.160526
    |  |           |2019-12-27    496.688779
    |  |           |2019-12-28    497.774215
2020|6 |OIL_2020_06|2020-03-24    500.899756
    |  |           |2020-03-25    500.830490
    |  |           |2020-03-26    501.829020
    |  |           |2020-03-27    501.875464
    |  |           |2020-03-28    503.241949
2020|9 |OIL_2020_09|2020-06-24    500.395880
    |  |           |2020-06-25    500.311780
    |  |           |2020-06-26    499.817331
    |  |           |2020-06-27    499.780468
    |  |           |2020-06-28    497.550235
...8 rows...
2021|6 |OIL_2021_06|2021-03-24    499.383712
    |  |           |2021-03-25    498.812289
    |  |           |2021-03-26    498.995159
    |  |           |2021-03-27    499.504985
    |  |           |2021-03-28    498.453581
2021|9 |OIL_2021_09|2021-06-24    499.310726
    |  |           |2021-06-25    500.248325
    |  |           |2021-06-26    500.458067
    |  |           |2021-06-27    498.482094
    |  |           |2021-06-28    499.704684
2021|12|OIL_2021_12|2021-09-24    501.359202
    |  |           |2021-09-25    501.178162
    |  |           |2021-09-26    501.846290
    |  |           |2021-09-27    502.217393
    |  |           |2021-09-28    500.996568
```

We observe that the cell, when recalculates, automatically caches the history and does not recalculate fake_ts. This is not magic. When a cell calculates its function, it provides the function with the variables it needs. Once calculated, it stores the output in **data** and will be able to provide **data** to the function next time, allowing it to avoid re-running expired calculations. Then cell will store the functions's result back in the **data** key for later use and this is repeated.

10.2.3 perdictable API

Parameters **on**, **renames** and **defaults** parameters determine the way the data is joined. If defaults is missing, the defaults from the function are used:

```
[45]: function = lambda price, quantity = 1: price * quantity
price = dictable(product = ['apple', 'banana', 'cherry'], price = [1,2,3])
quantity = dictable(product = ['apple', 'banana', 'damson'], quantity = [2,3,4])
perdictable(function, on = 'product')(price = price, quantity = quantity) ## cherry␣
↪should appear with default quantity
```

```
[45]: dictable[3 x 2]
product|data
apple  |2
banana |6
cherry |3
```

If you want to see the full calculations and inputs to the function set **include_inputs**=True:

```
[46]: perdictable(function, on = 'product', include_inputs = True)(price = price, quantity␣
↪= quantity)
```

```
[46]: dictable[3 x 5]
price|product|quantity|expiry|data
1    |apple  |2       |None  |2
2    |banana |3       |None  |6
3    |cherry |1       |None  |3
```

If you want output column to be not data, use **col**:

```
[47]: perdictable(function, on = 'product', include_inputs = True, col = 'cost')(price =␣
↪price, quantity = quantity)
```

```
[47]: dictable[3 x 5]
price|product|quantity|expiry|cost
1    |apple  |2       |None  |2
2    |banana |3       |None  |6
3    |cherry |1       |None  |3
```

The **if_none** parameter determines how data is calculated for rows that have expired but their data is None:

```
[52]: expiry = dictable(product = ['apple', 'banana', 'cherry'], expiry = [dt(-2), dt(-1),␣
↪dt(1)])
previous_data = dictable(product = ['apple', 'banana', 'cherry'], data = [None, 'some␣
↪value that will be kept', 'this value will be recalculated'])
perdictable(function, on = 'product', include_inputs = True, if_none = False)(price =␣
↪price, quantity = quantity, expiry = expiry, data = previous_data)
```

```
[52]: dictable[3 x 5]
expiry             |price|product|quantity|data
2021-02-24 00:00:00|1    |apple  |2       |None
2021-02-25 00:00:00|2    |banana |3       |some value that will be kept
2021-02-27 00:00:00|3    |cherry |1       |3
```

```
[53]: perdictable(function, on = 'product', include_inputs = True, if_none = True)(price =␣
↪price, quantity = quantity, expiry = expiry, data = previous_data)
```

```
[53]: dictable[3 x 5]
expiry             |price|product|quantity|data
```

(continues on next page)

(continued from previous page)

```
2021-02-24 00:00:00|1    |apple  |2       |2
2021-02-25 00:00:00|2    |banana |3       |some value that will be kept
2021-02-27 00:00:00|3    |cherry |1       |3
```

Some function want to receive historic data and they use it themselves. Parameter **output_is_input** controls this. For example: If your function is pulling historic prices from yahoo finance, you can use existing data to ask yahoo for only recent ones.

```
[54]: def running_total_costs(price, quantity=1, data=0):
          return data + price * quantity

      previous_data = dictable(product = ['apple', 'banana', 'cherry', 'damson'], data =␣
      ↪[10, 20, 30, 40])
      perdictable(running_total_costs, on = 'product', include_inputs = True)(price = price,
      ↪ quantity = quantity, data = previous_data)
```

```
[54]: dictable[3 x 5]
      price|product|quantity|expiry|data
      1    |apple  |2       |None  |12
      2    |banana |3       |None  |26
      3    |cherry |1       |None  |33
```

```
[57]: ## if you don't want existing data to be presented to the function:
      perdictable(running_total_costs, on = 'product', include_inputs = True, output_is_
      ↪input = False)(price = price, quantity = quantity, data = previous_data)
```

```
[57]: dictable[3 x 5]
      price|product|quantity|expiry|data
      1    |apple  |2       |None  |2
      2    |banana |3       |None  |6
      3    |cherry |1       |None  |3
```

10.3 Conclusions

pyg.base.join allows us to create joined table with the variables we need. This is leveraged by perdictable so that the ‘atomic’ data we work with is not a single timeseries but a whole table of timeseries data indexed by some keys. We can use various perdictable parameters to control cache policy. All this is done with very little additional code, allowing us to manage quite a lot of data items with very little effort while managing caching expired items.

CHAPTER

ELEVEN

PYG.TIMESERIES

Given pandas, why do we need this timeseries library? pandas is amazing but there are a few features in pyg.timeseries designed to enhance it.

There are three issues with pandas that pyg.timeseries tries to address:

- pandas works on pandas objects (obviously) but not on numpy arrays.
- pandas handles nan within timeseries inconsistently across its functions. This makes your results sensitive to reindexing/resampling. E.g.:
  - a.expanding() & a.ewm() **ignore** nan's for calculation and then forward-fill the result.
  - a.diff(), a.rolling() **include** any nans in the calculation, leading to nan propagation.
- pandas is great if you have the full timeseries. However, if you now want to run the same calculations in a live environment, on recent data, you have to append recent data at the end of the DataFrame and rerun.

pyg.timeseries tries to address this:

- pyg.timeseries agrees with pandas 100% (if there are no nan in the dataframe) while being of comparable speed
- pyg.timeseries works seemlessly on pandas objects and on numpy arrays, with no code change.
- pyg.timeseries handles nan consistently across all its functions, 'ignoring' all nan, making your results consistent regardless of reindexing/resampling.
- pyg.timeseries exposes the state of the internal function calculation. The exposure of internal state allows us to calculate the output of additional data **without** re-running history. This speeds up of two very common problems in finance:
  - risk calculations, Monte Carlo scenarios: We can run a trading strategy up to today and then generate multiple scenarios and see what-if, without having to rerun the full history.
  - live versus history: pandas is designed to run a full historical simulation. However, once we reach "today", speed is of the essense and running a full historical simulation every time we ingest a new price, is just too slow. That is why most fast trading is built around fast state-machines. Of course, making sure research & live versions do the same thing is tricky. pyg gives you the ability to run two systems in parallel with almost the same code base: run full history overnight and then run today's code base instantly, instantiated with the output of the historical simulation.

11.1 Agreement between pyg.timeseries and pandas

[1]:
```
from pyg import *; import pandas as pd; import numpy as np
s = pd.Series(np.random.normal(0,1,10000), drange(-9999)); a = s.values
t = pd.Series(np.random.normal(0,1,10000), drange(-9999))
```

[2]:
```
assert abs(s.count() - ts_count(s))< 1e-10
assert abs(s.mean() - ts_mean(s))  < 1e-10
assert abs(s.sum()  - ts_sum(s))   < 1e-10
assert abs(s.std()  - ts_std(s))   < 1e-10
assert abs(s.skew() - ts_skew(s))  < 1e-10
```

[3]:
```
assert abs(ewma(s, 10) - s.ewm(10).mean()).max()        < 1e-10
assert abs(ewmstd(s, 10) - s.ewm(10).std()).max()       < 1e-10
assert abs(ewmvar(s, 10) - s.ewm(10).var()).max()       < 1e-10
assert abs(ewmcor(s, t, 10) - s.ewm(10).corr(t)).max() < 1e-10
```

[4]:
```
assert abs(expanding_sum(s) - s.expanding().sum()).max()            < 1e-10
assert abs(expanding_mean(s) - s.expanding().mean()).max()          < 1e-10
assert abs(expanding_std(s) - s.expanding().std()).max()            < 1e-10
assert abs(expanding_skew(s) - s.expanding().skew()).max()          < 1e-10
assert abs(expanding_min(s) - s.expanding().min()).max()            < 1e-10
assert abs(expanding_max(s) - s.expanding().max()).max()            < 1e-10
assert abs(expanding_median(s) - s.expanding().median()).max()      < 1e-10
```

[5]:
```
assert abs(rolling_sum(s,10) - s.rolling(10).sum()).max()                            < 1e-
↪10
assert abs(rolling_mean(s,10) - s.rolling(10).mean()).max()                          < 1e-
↪10
assert abs(rolling_std(s,10) - s.rolling(10).std()).max()                            < 1e-
↪10
assert abs(rolling_skew(s,10) - s.rolling(10).skew()).max()                          < 1e-
↪10
assert abs(rolling_min(s,10) - s.rolling(10).min()).max()                            < 1e-
↪10
assert abs(rolling_max(s,10) - s.rolling(10).max()).max()                            < 1e-
↪10
assert abs(rolling_median(s,10) - s.rolling(10).median()).max()                      < 1e-
↪10
assert abs(rolling_quantile(s,10,0.3)[0.3] - s.rolling(10).quantile(0.3)).max()   < 1e-
↪10 ## The rolling_quantile returns the quantile as the header, since it supports␣
↪multiple quantiles calculations: e.g. rolling_quantile(s,10,[0.1,0.2,0.3,0.4,0.5,0.
↪6,0.7,0.8,0.9])
```

11.1.1 Quick performance comparison

pyg, when run on pandas dataframes rather than arrays, is of comparable speed to pandas

[6]:
```
compare = dictable(op =  ['rolling_sum', 'rolling_mean', 'rolling_std', 'rolling_min',
↪ 'rolling_median'],
             pyg = [rolling_sum, rolling_mean, rolling_std, rolling_min, rolling_
↪median],
             pandas = [s.rolling(10).sum, s.rolling(10).mean, s.rolling(10).std, s.
↪rolling(10).min, s.rolling(10).median]).do(lambda v: timer(v, n = 100, time = True),
↪ 'pyg', 'pandas')(pyg = lambda pyg: pyg(s, 10))(pandas = lambda pandas: pandas())
```

(continues on next page)

(continued from previous page)

```
compare += dictable(op =  ['expanding_sum', 'expanding_mean', 'expanding_std',
↪'expanding_min', 'expanding_median'],
             pyg = [expanding_sum, expanding_mean, expanding_std, expanding_min,␣
↪expanding_median],
             pandas = [s.expanding().sum, s.expanding().mean, s.expanding().std, s.
↪expanding().min, s.expanding().median]).do(lambda v: timer(v, n = 100, time = True),
↪ 'pyg', 'pandas')(pyg = lambda pyg: pyg(s))(pandas = lambda pandas: pandas())

compare += dictable(op =  ['ewma', 'ewmstd', 'ewmvar'],
             pyg = [ewma, ewmstd, ewmvar],
             pandas = [s.ewm(10).mean, s.ewm(10).std, s.ewm(10).var]).do(lambda v:␣
↪timer(v, n = 100, time = True), 'pyg', 'pandas')(pyg = lambda pyg: pyg(s,␣
↪10))(pandas = lambda pandas: pandas())

print(compare(winner = lambda pyg, pandas: 'pyg' if pyg<pandas * 0.8 else 'pandas' if␣
↪pyg > 1.2 * pandas else 'draw'))
```

```
2021-03-06 22:41:35,805 - pyg - INFO - TIMER:'rolling_sum' args:[["<class 'pandas.
↪core.series.Series'>[10000]", '10'], []] (100 runs) took 0:00:00.109934 sec
2021-03-06 22:41:35,898 - pyg - INFO - TIMER:'rolling_mean' args:[["<class 'pandas.
↪core.series.Series'>[10000]", '10'], []] (100 runs) took 0:00:00.087950 sec
2021-03-06 22:41:35,995 - pyg - INFO - TIMER:'rolling_std' args:[["<class 'pandas.
↪core.series.Series'>[10000]", '10'], []] (100 runs) took 0:00:00.090944 sec
2021-03-06 22:41:36,116 - pyg - INFO - TIMER:'rolling_min' args:[["<class 'pandas.
↪core.series.Series'>[10000]", '10'], []] (100 runs) took 0:00:00.119930 sec
2021-03-06 22:41:36,269 - pyg - INFO - TIMER:'rolling_median' args:[["<class 'pandas.
↪core.series.Series'>[10000]", '10'], []] (100 runs) took 0:00:00.153483 sec
2021-03-06 22:41:36,351 - pyg - INFO - TIMER:'sum' args:[[], []] (100 runs) took 0:00:
↪00.079685 sec
2021-03-06 22:41:36,465 - pyg - INFO - TIMER:'mean' args:[[], []] (100 runs) took 0:
↪00:00.112599 sec
2021-03-06 22:41:36,585 - pyg - INFO - TIMER:'std' args:[[], []] (100 runs) took 0:00:
↪00.119877 sec
2021-03-06 22:41:36,687 - pyg - INFO - TIMER:'min' args:[[], []] (100 runs) took 0:00:
↪00.100942 sec
2021-03-06 22:41:37,378 - pyg - INFO - TIMER:'median' args:[[], []] (100 runs) took 0:
↪00:00.688107 sec
2021-03-06 22:41:37,467 - pyg - INFO - TIMER:'expanding_sum' args:[["<class 'pandas.
↪core.series.Series'>[10000]"], []] (100 runs) took 0:00:00.086951 sec
2021-03-06 22:41:37,528 - pyg - INFO - TIMER:'expanding_mean' args:[["<class 'pandas.
↪core.series.Series'>[10000]"], []] (100 runs) took 0:00:00.059966 sec
2021-03-06 22:41:37,651 - pyg - INFO - TIMER:'expanding_std' args:[["<class 'pandas.
↪core.series.Series'>[10000]"], []] (100 runs) took 0:00:00.120938 sec
2021-03-06 22:41:37,695 - pyg - INFO - TIMER:'expanding_min' args:[["<class 'pandas.
↪core.series.Series'>[10000]"], []] (100 runs) took 0:00:00.040991 sec
2021-03-06 22:41:37,903 - pyg - INFO - TIMER:'expanding_median' args:[["<class
↪'pandas.core.series.Series'>[10000]"], []] (100 runs) took 0:00:00.206892 sec
2021-03-06 22:41:37,939 - pyg - INFO - TIMER:'sum' args:[[], []] (100 runs) took 0:00:
↪00.033967 sec
2021-03-06 22:41:38,002 - pyg - INFO - TIMER:'mean' args:[[], []] (100 runs) took 0:
↪00:00.059981 sec
2021-03-06 22:41:38,075 - pyg - INFO - TIMER:'std' args:[[], []] (100 runs) took 0:00:
↪00.071959 sec
2021-03-06 22:41:38,245 - pyg - INFO - TIMER:'min' args:[[], []] (100 runs) took 0:00:
↪00.168553 sec
2021-03-06 22:41:39,523 - pyg - INFO - TIMER:'median' args:[[], []] (100 runs) took 0:
↪00:01.277246 sec
```

(continues on next page)

(continued from previous page)

```
2021-03-06 22:41:39,620 - pyg - INFO - TIMER:'ewma' args:[["<class 'pandas.core.
↪series.Series'>[10000]", '10'], []] (100 runs) took 0:00:00.094945 sec
2021-03-06 22:41:39,732 - pyg - INFO - TIMER:'ewmstd' args:[["<class 'pandas.core.
↪series.Series'>[10000]", '10'], []] (100 runs) took 0:00:00.110924 sec
2021-03-06 22:41:39,827 - pyg - INFO - TIMER:'ewmvar' args:[["<class 'pandas.core.
↪series.Series'>[10000]", '10'], []] (100 runs) took 0:00:00.093965 sec
2021-03-06 22:41:39,855 - pyg - INFO - TIMER:'mean' args:[[], []] (100 runs) took 0:
↪00:00.026971 sec
2021-03-06 22:41:39,953 - pyg - INFO - TIMER:'std' args:[[], []] (100 runs) took 0:00:
↪00.096954 sec
2021-03-06 22:41:39,995 - pyg - INFO - TIMER:'var' args:[[], []] (100 runs) took 0:00:
↪00.039983 sec
```

```
op              |pandas        |pyg           |winner
rolling_sum     |0:00:00.079685|0:00:00.109934|pandas
rolling_mean    |0:00:00.112599|0:00:00.087950|pyg
rolling_std     |0:00:00.119877|0:00:00.090944|pyg
rolling_min     |0:00:00.100942|0:00:00.119930|draw
rolling_median  |0:00:00.688107|0:00:00.153483|pyg
expanding_sum   |0:00:00.033967|0:00:00.086951|pandas
expanding_mean  |0:00:00.059981|0:00:00.059966|draw
expanding_std   |0:00:00.071959|0:00:00.120938|pandas
expanding_min   |0:00:00.168553|0:00:00.040991|pyg
expanding_median|0:00:01.277246|0:00:00.206892|pyg
ewma            |0:00:00.026971|0:00:00.094945|pandas
ewmstd          |0:00:00.096954|0:00:00.110924|draw
ewmvar          |0:00:00.039983|0:00:00.093965|pandas
```

11.2 pyg and numpy arrays

pyg supports numpy arrays natively. Indeed, pyg is 3-5 times faster on numpy arrays.

[7]:
```python
a = s.values
assert abs(ts_count(a) - ts_count(s))< 1e-10
assert abs(ts_mean(a) - ts_mean(s))  < 1e-10
assert abs(ts_sum(a)  - ts_sum(s))   < 1e-10
assert abs(ts_std(a)  - ts_std(s))   < 1e-10
assert abs(ts_skew(a) - ts_skew(s))  < 1e-10
```

[8]:
```python
assert abs(ewma(s, 10) - ewma(a,10)).max()                       < 1e-10
assert abs(ewmstd(s, 10) - ewmstd(a,10)).max()                   < 1e-10
assert abs(ewmvar(s, 10) - ewmvar(a,10)).max()                   < 1e-10
assert abs(ewmcor(s, t, 10) - ewmcor(a, t.values, 10)).max()   < 1e-10
```

[9]:
```python
assert abs(expanding_sum(s) - expanding_sum(a)).max()                 < 1e-10
assert abs(expanding_min(s) - expanding_min(a)).max()                 < 1e-10
assert abs(expanding_max(s) - expanding_max(a)).max()                 < 1e-10
assert abs(expanding_mean(s) - expanding_mean(a)).max()               < 1e-10
assert abs(expanding_std(s) - expanding_std(a)).max()                 < 1e-10
assert abs(expanding_skew(s) - expanding_skew(a)).max()               < 1e-10
assert abs(expanding_median(s) - expanding_median(a)).max()           < 1e-10
```

[10]:
```
assert abs(rolling_sum(s,10) - rolling_sum(a,10)).max()               < 1e-10
assert abs(rolling_min(s,10) - rolling_min(a,10)).max()               < 1e-10
assert abs(rolling_max(s,10) - rolling_max(a,10)).max()               < 1e-10
assert abs(rolling_mean(s,10) - rolling_mean(a,10)).max()             < 1e-10
assert abs(rolling_std(s,10) - rolling_std(a,10)).max()               < 1e-10
assert abs(rolling_skew(s,10) - rolling_skew(a,10)).max()             < 1e-10
assert abs(rolling_median(s,10) - rolling_median(a,10)).max()         < 1e-10
```

11.3 pandas treatment of nan

Suppose we have weekly data that at some point we resample to daily… The two look the same…

[11]:
```
t0 = dt_bump('20210301', '-999w')
days = drange(t0,'20210301','1b')
weekly = pd.Series(np.random.normal(0,1,1000), drange(t0,None,'1w')); weekly.name =
↪'weekly'
daily = weekly.reindex(days); daily.name = 'daily'
pd.concat([weekly,daily], axis = 1)
```

[11]:
```
              weekly     daily
2002-01-07  0.423187  0.423187
2002-01-08       NaN       NaN
2002-01-09       NaN       NaN
2002-01-10       NaN       NaN
2002-01-11       NaN       NaN
...              ...       ...
2021-02-23       NaN       NaN
2021-02-24       NaN       NaN
2021-02-25       NaN       NaN
2021-02-26       NaN       NaN
2021-03-01  1.408439  1.408439

[4996 rows x 2 columns]
```

… but any calculation using the daily will yield a different result from a calculation on the weekly which is then resampled to daily:

[12]:
```
pd.concat([weekly.ewm(4).mean().reindex(days), daily.ewm(4).mean()], axis = 1) ## The␣
↪result depends on what is done first...
```

[12]:
```
              weekly     daily
2002-01-07  0.423187  0.423187
2002-01-08       NaN  0.423187
2002-01-09       NaN  0.423187
2002-01-10       NaN  0.423187
2002-01-11       NaN  0.423187
...              ...       ...
2021-02-23       NaN  0.178687
2021-02-24       NaN  0.178687
2021-02-25       NaN  0.178687
2021-02-26       NaN  0.178687
2021-03-01  0.655222  1.005474

[4996 rows x 2 columns]
```

[13]:
```
pd.concat([weekly.diff().reindex(days), daily.diff()], axis = 1) ## The result␣
↪depends on what is done first...
```

[13]:
```
              weekly  daily
2002-01-07       NaN    NaN
2002-01-08       NaN    NaN
2002-01-09       NaN    NaN
2002-01-10       NaN    NaN
2002-01-11       NaN    NaN
...              ...    ...
2021-02-23       NaN    NaN
2021-02-24       NaN    NaN
2021-02-25       NaN    NaN
2021-02-26       NaN    NaN
2021-03-01  1.644159    NaN

[4996 rows x 2 columns]
```

Indeed, for diff, daily.diff() is all nan!

11.4 pyg.timeseries treatment of nans

pyg treats nan as if they are not there, so the fact that we resampled the data and introduced lots of nan's does not affect the calculations. We find this to be a more logical (and less error prone) approach.

[14]:
```
nona(pd.concat([ewma(weekly, 4).reindex(days), ewma(daily,4)], axis = 1)) ## The two␣
↪match exactly
```

[14]:
```
                   0         1
2002-01-07  0.423187  0.423187
2002-01-14 -0.105302 -0.105302
2002-01-21 -0.019371 -0.019371
2002-01-28  0.332137  0.332137
2002-02-04  0.559419  0.559419
...              ...       ...
2021-02-01  0.369931  0.369931
2021-02-08  0.526351  0.526351
2021-02-15  0.642578  0.642578
2021-02-22  0.466918  0.466918
2021-03-01  0.655222  0.655222

[1000 rows x 2 columns]
```

[15]:
```
nona(pd.concat([diff(weekly).reindex(days), diff(daily)], axis = 1)) ## The result␣
↪depends on what is done first...
```

[15]:
```
                   0         1
2002-01-14 -0.951280 -0.951280
2002-01-21  0.632462  0.632462
2002-01-28  0.913911  0.913911
2002-02-04  0.077887  0.077887
2002-02-11 -2.180086 -2.180086
...              ...       ...
2021-02-01 -0.678079 -0.678079
2021-02-08  1.049093  1.049093
2021-02-15 -0.044543 -0.044543
```

(continues on next page)

(continued from previous page)

```
2021-02-22 -1.343206 -1.343206
2021-03-01  1.644159  1.644159

[999 rows x 2 columns]
```

11.5 Using pyg.timeseries to manage state

One of the problem in timeseries analysis is writing research code that works in analysing past data but ideally, the same code can be used in live application. One easy approach is "stick the extra data point at the end and run it again from 1980". This leaves us with a single code base but for many live applications (e.g. live trading), this is not viable.

Further, given our positions today, we may want to run simulations of "what happens next?" to understand what the system is likely to do should various events occur. Risk calculations are expensive and re-running 10k Monte Carlo scenarios, each time running from 1980 is expensive.

Conversely, we can run research and live systems on two separate code base. This makes live systems responsive but six months down the line, we realise research code base and live code base did not do quite the same thing.

pyg approaches this problem by exposing the internal state of each of its calculation. Each function has two versions:

- function(. . .) returns the calculation as performed by pandas
- function_(. . .) returns a dictionary of dict(data = , state =). The data agrees with function(. . .) while the state is a dict we can instantiate new calculations with.

[16]:
```
from pyg import *
history = pd.Series(np.random.normal(0,1,1000), drange(-1000,-1))
history_signal = ewma_(history, 10)
history_signal # The output consists of 'data' and 'state' where data matches a␣
↪normal ewma calculation
```

[16]:
```
{'data': 2018-06-10   -0.511500
 2018-06-11    0.445609
 2018-06-12   -0.065606
 2018-06-13   -0.358735
 2018-06-14   -0.069188
                 ...
 2021-03-01   -0.144503
 2021-03-02   -0.066708
 2021-03-03   -0.141431
 2021-03-04   -0.122797
 2021-03-05   -0.051610
 Length: 1000, dtype: float64,
 'state': {'t': nan, 't0': 0.9999999999999994, 't1': -0.05161000819451757}}
```

[17]:
```
live = pd.Series(np.random.normal(0,1,10), drange(9))
live_signal = ewma(live, 10, state = history_signal.state) ## I only feed in live␣
↪timeseries
'live: from today onwards', live_signal
```

[17]:
```
('live: from today onwards',
 2021-03-06   -0.059815
 2021-03-07   -0.165151
 2021-03-08   -0.104525
 2021-03-09   -0.160978
 2021-03-10   -0.224791
```

(continues on next page)

(continued from previous page)

```
2021-03-11   -0.325723
2021-03-12   -0.207468
2021-03-13   -0.233642
2021-03-14   -0.228141
2021-03-15   -0.244483
dtype: float64)
```

[18]:
```
joint_data = pd.concat([history, live])
joint_signal = ewma(joint_data, 10)
assert eq(live_signal, joint_signal[dt(0):])  # The live signal is the same, even␣
↪though it only received live data for its calculation.
joint_signal[dt(0):]
```

[18]:
```
2021-03-06   -0.059815
2021-03-07   -0.165151
2021-03-08   -0.104525
2021-03-09   -0.160978
2021-03-10   -0.224791
2021-03-11   -0.325723
2021-03-12   -0.207468
2021-03-13   -0.233642
2021-03-14   -0.228141
2021-03-15   -0.244483
dtype: float64
```

This allows us to set up three parallel pipelines that share a virtually identical codebase:

workflow	historic data	live data	risk analysis
when run?	research/overnight	live	overnight
data source?	ts = long timeseries	a = short ts/array	1000's of sims
speed?	slow, non-critical	instantenous	quick
apply f to data	x_ = f_(ts)	x = f(a, **x_)	same as live
apply g	y_ = g_(ts, **x_**)	y = g(a, x, **y_)	same as live
final result h	z_ = h_(ts, **x_**, **y_**)	z = h(a, x, y, **z_)	same as live

Note that for live trading or risk analysis, we tend to switch and run on numpy arrays rather than pandas object. This speeds up the calculations while introduces no code change. In the example below we explore how to create state-aware, functions within pyg. The paradigm is that for most functions, function_ will return not just the timeseries output but also the states.

11.5.1 Example: creating a function exposing its state

Suppose we try to write an ewma crossover function (the difference of two ewma). We want to normalize it by its own volatility. Traditionally we will write:

[19]:
```
def pandas_crossover(a, fast, slow, vol):
    fast_ewma = a.ewm(fast).mean()
    slow_ewma = a.ewm(slow).mean()
    raw_signal = fast_ewma - slow_ewma
    signal_rms = (raw_signal**2).ewm(vol).mean()**0.5
    signal_rms[signal_rms==0] = np.nan
    normalized = raw_signal/signal_rms
    return normalized
```

(continues on next page)

(continued from previous page)

```
a = pd.Series(np.random.normal(0,1,10000), drange(-9999)); fast = 10; slow = 30; vol␣
↪= 50
pandas_x = pandas_crossover(a, fast, slow, vol)
pandas_x
```

[19]:
```
1993-10-20         NaN
1993-10-21   -1.407264
1993-10-22   -1.714259
1993-10-23    1.177760
1993-10-24   -1.220600
                ...
2021-03-02   -1.767405
2021-03-03   -1.183420
2021-03-04   -1.764486
2021-03-05   -2.458497
2021-03-06   -2.242366
Length: 10000, dtype: float64
```

We can quickly rewrite it using pyg:

[28]:
```
def crossover(a, fast, slow, vol):
    fast_ewma = ewma(a, fast)
    slow_ewma = ewma(a, slow)
    raw_signal = fast_ewma - slow_ewma
    signal_rms = ewmrms(raw_signal, vol)
    signal_rms = v2na(signal_rms)
    normalized = raw_signal/signal_rms
    return normalized
x = crossover(a, fast, slow, vol)
assert abs(x-pandas_x).max()<1e-10
x
```

[28]:
```
1993-10-20   -1.000000
1993-10-21   -1.407264
1993-10-22   -1.714259
1993-10-23    1.177760
1993-10-24   -1.220600
                ...
2021-03-02   -1.767405
2021-03-03   -1.183420
2021-03-04   -1.764486
2021-03-05   -2.458497
2021-03-06   -2.242366
Length: 10000, dtype: float64
```

And with very little additional effort, we can write a new function that also exposes the internal state:

[29]:
```
_data = 'data'
def crossover_(a, fast, slow, vol, instate = None):
    state = Dict(fast = {}, slow = {}, vol = {}) if instate is None else instate
    fast_ewma_ = ewma_(a, fast, instate = state.fast)
    slow_ewma_ = ewma_(a, slow, instate = state.slow)
    raw_signal = fast_ewma_.data - slow_ewma_.data
    signal_rms = ewmrms_(raw_signal, vol, instate = state.vol)
    normalized = raw_signal/v2na(signal_rms.data)
    return Dict(data = normalized, state = Dict(fast = fast_ewma_.state, slow = slow_
↪ewma_.state, vol = signal_rms.state))
```

(continues on next page)

(continued from previous page)

```
crossover_.output = ['data', 'state'] # output declares the function to have a dict␣
↪output and is used by cell

def crossover(a, fast, slow, vol, state = None):
    return crossover_(a, fast, slow, vol, instate = state).data

x_ = crossover_(a, fast, slow, vol)
assert eq(x, x_.data) and eq(x, crossover(a, fast, slow, vol))
x_.data
```

[29]:
```
1993-10-20   -1.000000
1993-10-21   -1.407264
1993-10-22   -1.714259
1993-10-23    1.177760
1993-10-24   -1.220600
                ...
2021-03-02   -1.767405
2021-03-03   -1.183420
2021-03-04   -1.764486
2021-03-05   -2.458497
2021-03-06   -2.242366
Length: 10000, dtype: float64
```

The three give idential results and we can also verify that crossover_ will allow us to split the evaluation to the long-history and the new data:

[45]:
```
history = a[:9900]
live = a[9900:].values
x_history = crossover_(history, 10, 30, 50)
x_live = crossover(live, 10, 30, 50, state = x_history.state)
x_ = crossover_(a, fast, slow, vol)
assert eq(x_live , x_.data[9900:].values)
```

Have we gained anything?

[46]:
```
pandas_old = timer(pandas_crossover, 100, time = True)(history, 10, 30, 50)
x_history  = crossover_(history, 10, 30, 50)
x_history_time  = timer(crossover_, 100, time = True)(history, 10, 30, 50)
x_live = timer(crossover, 100, time = True)(live, 10, 30, 50, state = x_history.state)
'pandas: ', pandas_old.microseconds//1000, 'pyg history:', x_history_time.
↪microseconds//1000, 'pyg_live:', x_live.microseconds//1000
```

```
2021-03-06 23:55:39,746 - pyg - INFO - TIMER:'pandas_crossover' args:[["<class
↪'pandas.core.series.Series'>[9900]", '10', '30', '50'], []] (100 runs) took 0:00:00.
↪373514 sec
2021-03-06 23:55:39,953 - pyg - INFO - TIMER:'crossover_' args:[["<class 'pandas.core.
↪series.Series'>[9900]", '10', '30', '50'], []] (100 runs) took 0:00:00.202883 sec
2021-03-06 23:55:40,004 - pyg - INFO - TIMER:'crossover' args:[["<class 'numpy.ndarray
↪'>[100]", '10', '30', '50'], ["state=<class 'pyg.base._dict.Dict'>[3]"]] (100 runs)␣
↪took 0:00:00.049972 sec
```

[46]:
```
('pandas: ', 373, 'pyg history:', 202, 'pyg_live:', 49)
```

We see that pyg is already faster than pandas. Running just the new data using numpy arrays, is about 4-5 times faster still. Indeed, running 10k 100-day forward scenarios take about 2 seconds at most.

[48]:
```
scenarios = np.random.normal(0,1,(100,10000))
x_scenarios = timer(crossover)(scenarios , 10, 30, 50, state = x_history.state)
```

```
2021-03-06 23:56:10,252 - pyg - INFO - TIMER:'crossover' args:[["<class 'numpy.ndarray
↪'>[100]", '10', '30', '50'], ["state=<class 'pyg.base._dict.Dict'>[3]"]] (1 runs)␣
↪took 0:00:01.605710 sec
```

Using cells, our code looks like this, with live and historical codebase looking pretty similar

[49]:
```
x_history = cell(crossover_, a = history, fast = 10, slow = 30, vol = 50)()
x_live = cell(crossover, a = live, fast = 10, slow = 30, vol = 50, state = x_
↪history)()
x_history
```

[49]:
```
cell
a:
    1993-10-20    0.463739
    1993-10-21    0.429161
    1993-10-22   -0.342095
    1993-10-23    1.192557
    1993-10-24   -0.448828
                    ...
    2020-11-22   -0.272184
    2020-11-23    0.121197
    2020-11-24   -0.581223
    2020-11-25   -0.682961
    2020-11-26   -1.084583
    Length: 9900, dtype: float64
fast:
    10
slow:
    30
vol:
    50
function:
    <function crossover_ at 0x000001CF9B58BA60>
instate:
    None
data:
    1993-10-20   -1.000000
    1993-10-21   -1.407264
    1993-10-22   -1.714259
    1993-10-23    1.177760
    1993-10-24   -1.220600
                    ...
    2020-11-22   -2.091785
    2020-11-23   -1.765958
    2020-11-24   -1.796933
    2020-11-25   -1.853106
    2020-11-26   -2.044795
    Length: 9900, dtype: float64
state:
    Dict
    fast:
        {'t': nan, 't0': 0.9999999999999994, 't1': -0.4251894284980144}
    slow:
        {'t': nan, 't0': 0.9999999999999983, 't1': -0.14408421908740027}
    vol:
        {'t': nan, 't0': 0.9999999999999972, 't2': 0.01889897942675779}
```

[50]:
```
pd.concat([pd.Series(x_live.data, pandas_x.index[-100:]), pandas_x.iloc[-100:]], axis
↪= 1)
```

[50]:
```
                   0         1
2020-11-27 -2.466036 -2.466036
2020-11-28 -1.899795 -1.899795
2020-11-29 -1.573653 -1.573653
2020-11-30 -1.473624 -1.473624
2020-12-01 -1.978180 -1.978180
...              ...       ...
2021-03-02 -1.767405 -1.767405
2021-03-03 -1.183420 -1.183420
2021-03-04 -1.764486 -1.764486
2021-03-05 -2.458497 -2.458497
2021-03-06 -2.242366 -2.242366

[100 rows x 2 columns]
```

CHAPTER

TWELVE

PYG.TIMESERIES DECORATORS

There are a few decorators that are relevant to timeseries analysis ## pd2np and compiled We write most of our underlying functions assuming the function parameters are 1-d numpy arrays. If you want them numba.jit compiled, please use the compiled operator.

[1]:
```
from pyg import *
import pandas as pd; import numpy as np
@pd2np
@compiled
def sumsq(a, total = 0.0):
    res = np.empty_like(a)
    for i in range(a.shape[0]):
        if np.isnan(a[i]):
            res[i] = np.nan
        else:
            total += a[i]**2
            res[i] = total
    return res
```

It is not surpising that sumsq works for arrays. Notice how np.isnan is handled to ensure nans are skipped.

[2]:
```
a = np.arange(5)
sumsq(a)
```

[2]:
```
array([ 0,  1,  5, 14, 30])
```

pd2np will convert a pandas Series to arrays, run the function and convert back to pandas. This will only work for a 1-dimensional objects, so no df nor 2-d np.ndarray.

[3]:
```
s = pd.Series(a, drange(-4))
sumsq(s)
```

[3]:
```
2021-02-27     0
2021-02-28     1
2021-03-01     5
2021-03-02    14
2021-03-03    30
dtype: int32
```

12.1 loop

We decorate sumsq with the **loop** decorator. Once we introduce loop, The function will loop over columns of a DataFrame or a numpy array:

```
[4]: @loop(pd.DataFrame, dict, list, np.ndarray)
@pd2np
@compiled
def sumsq(a, total = 0):
    res = np.empty_like(a)
    for i in range(a.shape[0]):
        if np.isnan(a[i]):
            res[i] = np.nan
        else:
            total += a[i]**2
            res[i] = total
    return res

df = pd.DataFrame(dict(a = a, b = a+1), drange(-4))
df
```

```
[4]:
            a  b
2021-02-27  0  1
2021-02-28  1  2
2021-03-01  2  3
2021-03-02  3  4
2021-03-03  4  5
```

```
[5]: sumsq(df)
```

```
[5]:
             a   b
2021-02-27   0   1
2021-02-28   1   5
2021-03-01   5  14
2021-03-02  14  30
2021-03-03  30  55
```

Indeed, since we asked it to loop over dict, list and numpy array (2d)

```
[6]: sumsq(dict(a = a, b = a+1))
```

```
[6]: {'a': array([ 0,  1,  5, 14, 30]), 'b': array([ 1,  5, 14, 30, 55])}
```

```
[7]: sumsq(df.values)
```

```
[7]: array([[ 0,  1],
       [ 1,  5],
       [ 5, 14],
       [14, 30],
       [30, 55]])
```

12.2 presync: manage indexing and date stamps

Suppose the function takes two (or more) timeseries.

[8]:
```
@presync(index = 'inner')
@loop(pd.DataFrame, np.ndarray)
@pd2np
def product(a, b):
    return a * b
```

[9]:
```
a = np.arange(5); b = np.arange(5)
product(a,b)
```

[9]:
```
array([ 0,  1,  4,  9, 16])
```

What happens when the weights and the timeseries are unsynchronized?

[10]:
```
a_ = pd.Series(a, drange(-4)) ; a_.name = 'a'
b_ = pd.Series(b, drange(-3,1)); b_.name = 'b'
pd.concat([a_, b_], axis=1)
```

[10]:
```
              a    b
2021-02-27  0.0  NaN
2021-02-28  1.0  0.0
2021-03-01  2.0  1.0
2021-03-02  3.0  2.0
2021-03-03  4.0  3.0
2021-03-04  NaN  4.0
```

[11]:
```
product(a_, b_) ## just the inner values
```

[11]:
```
2021-02-28     0
2021-03-01     2
2021-03-02     6
2021-03-03    12
Freq: D, dtype: int32
```

[12]:
```
product.oj(a_, b_) ## outer join
```

[12]:
```
2021-02-27     NaN
2021-02-28     0.0
2021-03-01     2.0
2021-03-02     6.0
2021-03-03    12.0
2021-03-04     NaN
Freq: D, dtype: float64
```

[13]:
```
product.oj.ffill(a_, b_) ## outer join and forward-fill
```

[13]:
```
2021-02-27     NaN
2021-02-28     0.0
2021-03-01     2.0
2021-03-02     6.0
2021-03-03    12.0
2021-03-04    16.0
Freq: D, dtype: float64
```

12.2.1 presync and numpy arrays

When we deal with thousands of equities, one way of speeding calculations is by stacking them all onto huge dataframes. This does work but one is always busy fiddling with 'the universe' one is trading. We took a slightly different approach:

- We define a global timestamp.
- We then sample each timeseries to that global timestamp, dropping the early history where the data is all nan. (df_fillna(ts, index, method = 'fnna')).
- We then do our research on these numpy arrays.
- Finally, once we are done, we resample back to the global timestamp.

While we are in numpy arrays, we can 'inner join' by recognising the 'end' of each array shares the same date. Indeed df_index, df_reindex and presync all work seemlessly on np.ndarray as well as DataFrames, under that assumption that **the end of all arrays are in sync**.

We find this approach saves on memory and on computation time. It also lends itself to being able to retrieve and create specific universes for specific trading ideas. It is not without its own issues but that is a separate discussion.

[14]:
```
a = np.arange(5); b = np.arange(1,5)
a, b
```

[14]:
```
(array([0, 1, 2, 3, 4]), array([1, 2, 3, 4]))
```

[15]:
```
product(a, b)
```

[15]:
```
array([ 1,  4,  9, 16])
```

[16]:
```
us = calendar('US')
dates = pd.Index(us.drange('-40y', 0 ,'1b'))
```

[17]:
```
universe = dictable(stock = ['msft', 'appl', 'tsla'], n = [10000, 8000, 7000])
universe = universe(ts = lambda n: pd.Series(np.random.normal(0,1,n+1), us.drange('-
↪%ib'%n, 0, '1b'))[np.random.normal(0,1,n+1)>-1])
universe
```

[17]:
```
dictable[3 x 3]
stock|n    |ts
msft |10000|1982-11-03   -1.309868
     |     |1982-11-04   -0.737816
     |     |1982-11-05    0.460173
     |     |1982-11-08   -0.895898
     |     |1982-11-09   -0.813305
appl |8000 |1990-07-04    0.040855
     |     |1990-07-05   -1.327995
     |     |1990-07-06    0.114328
     |     |1990-07-09   -1.626176
     |     |1990-07-10   -0.031428
tsla |7000 |1994-05-04   -1.259911
     |     |1994-05-05    1.014304
     |     |1994-05-09   -0.035104
     |     |1994-05-10   -1.265964
     |     |1994-05-11   -0.001664
```

[18]:
```
universe = universe(rtn = lambda ts: ts.values)
universe = universe(price = lambda rtn : cumsum(rtn))
```

(continues on next page)

(continued from previous page)

```
universe = universe(vol = lambda rtn: ewmstd(rtn, 30))
universe
```

[18]:
```
dictable[3 x 6]
stock|n    |ts                    |rtn                                                     ␣
↪|price                                                 |vol
msft |10000|1982-11-03   -1.309868|[-1.3098679  -0.73781612  0.4601727  ... -0.
↪327291|[-1.3098679  -2.04768402 -1.58751132 ...  4.750977|[        nan         nan  ␣
↪    nan ... 1.02923517 1
     |     |1982-11-04   -0.737816|  0.67289106]                                        ␣
↪|  5.89220017]                                         |
     |     |1982-11-05    0.460173|                                                    ␣
↪|                                                      |
     |     |1982-11-08   -0.895898|                                                    ␣
↪|                                                      |
     |     |1982-11-09   -0.813305|                                                    ␣
↪|                                                      |
appl |8000 |1990-07-04    0.040855|[ 0.04085499 -1.32799499  0.11432766 ... -1.
↪017795|[ 4.08549924e-02 -1.28714000e+00 -1.17281234e+00 .|[        nan         nan  ␣
↪    nan ... 0.88535052 0
     |     |1990-07-05   -1.327995| -0.82540937]                                        ␣
↪|  7.67908570e+01  7.59654476e+01]                     |
     |     |1990-07-06    0.114328|                                                    ␣
↪|                                                      |
     |     |1990-07-09   -1.626176|                                                    ␣
↪|                                                      |
     |     |1990-07-10   -0.031428|                                                    ␣
↪|                                                      |
tsla |7000 |1994-05-04   -1.259911|[-1.25991126  1.01430418 -0.0351036  ... -0.
↪174814|[-1.25991126 -0.24560708 -0.28071068 ... 24.331768|[        nan         nan  ␣
↪    nan ... 0.94944115 0
     |     |1994-05-05    1.014304| -0.69279468]                                        ␣
↪| 23.93415613]                                         |
     |     |1994-05-09   -0.035104|                                                    ␣
↪|                                                      |
     |     |1994-05-10   -1.265964|                                                    ␣
↪|                                                      |
     |     |1994-05-11   -0.001664|                                                    ␣
↪|                                                      |
```

[19]:
```
presync(lambda tss: np.array(tss).T)(universe.vol)
```

[19]:
```
array([[1.01584217, 0.95105069,        nan],
       [1.02939552, 0.99139701,        nan],
       [1.01323584, 0.97982437,        nan],
       ...,
       [1.02923517, 0.88535052, 0.94944115],
       [1.018515  , 0.91252795, 0.93434464],
       [1.01216505, 0.91053212, 0.93155713]])
```

[20]:
```
universe = universe.do(lambda value: np_reindex(value, dates), 'rtn', 'price', 'vol')
universe
```

[20]:
```
dictable[3 x 6]
stock|n    |ts                    |rtn                   |price                 |vol
msft |10000|1982-11-03   -1.309868|1988-11-21   -1.309868|1988-11-21   -1.309868␣
↪|1988-11-21         NaN
```

(continues on next page)

(continued from previous page)

```
     |     |1982-11-04   -0.737816|1988-11-22   -0.737816|1988-11-22   -2.047684␣
↪|1988-11-22         NaN
     |     |1982-11-05    0.460173|1988-11-23    0.460173|1988-11-23   -1.587511␣
↪|1988-11-23         NaN
     |     |1982-11-08   -0.895898|1988-11-24   -0.895898|1988-11-24   -2.483409␣
↪|1988-11-24         NaN
     |     |1982-11-09   -0.813305|1988-11-25   -0.813305|1988-11-25   -3.296714␣
↪|1988-11-25         NaN
appl |8000 |1990-07-04    0.040855|1995-04-20    0.040855|1995-04-20     0.
↪040855|1995-04-20         NaN
     |     |1990-07-05   -1.327995|1995-04-21   -1.327995|1995-04-21    -1.
↪287140|1995-04-21         NaN
     |     |1990-07-06    0.114328|1995-04-24    0.114328|1995-04-24    -1.
↪172812|1995-04-24         NaN
     |     |1990-07-09   -1.626176|1995-04-25   -1.626176|1995-04-25    -2.
↪798988|1995-04-25         NaN
     |     |1990-07-10   -0.031428|1995-04-26   -0.031428|1995-04-26    -2.
↪830417|1995-04-26         NaN
tsla |7000 |1994-05-04   -1.259911|1998-09-11   -1.259911|1998-09-11    -1.
↪259911|1998-09-11         NaN
     |     |1994-05-05    1.014304|1998-09-14    1.014304|1998-09-14    -0.
↪245607|1998-09-14         NaN
     |     |1994-05-09   -0.035104|1998-09-15   -0.035104|1998-09-15    -0.
↪280711|1998-09-15         NaN
     |     |1994-05-10   -1.265964|1998-09-16   -1.265964|1998-09-16    -1.
↪546674|1998-09-16         NaN
     |     |1994-05-11   -0.001664|1998-09-17   -0.001664|1998-09-17    -1.
↪548338|1998-09-17         NaN
```

[21]:
```
vol = pd.concat(universe.vol, axis = 1); vol.columns = universe.stock
vol
```

[21]:
```
                msft      appl      tsla
1988-11-21       NaN       NaN       NaN
1988-11-22       NaN       NaN       NaN
1988-11-23       NaN       NaN       NaN
1988-11-24       NaN       NaN       NaN
1988-11-25       NaN       NaN       NaN
...              ...       ...       ...
2021-02-25  1.063016  0.890791  0.931185
2021-02-26  1.045735  0.880376  0.963182
2021-03-01  1.029235  0.885351  0.949441
2021-03-02  1.018515  0.912528  0.934345
2021-03-03  1.012165  0.910532  0.931557

[8423 rows x 3 columns]
```

CHAPTER

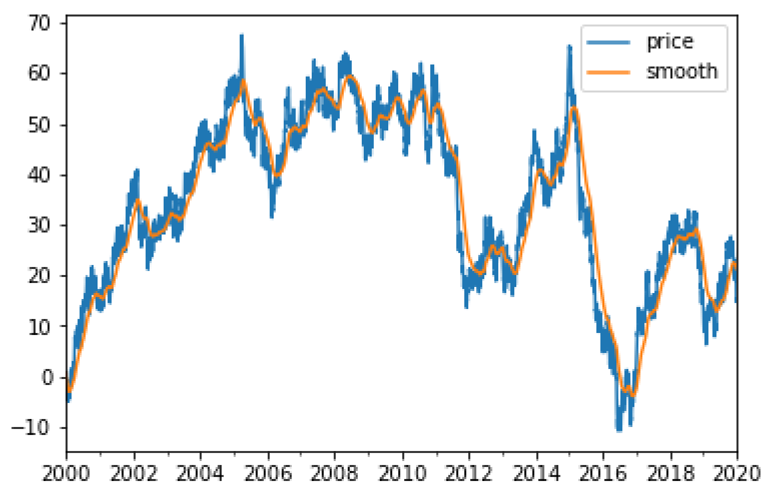
THIRTEEN

PYG.TIMESERIES.EWMA

The ewm functions implement the concept of time which we think is worthwhile explaining. We start with an example

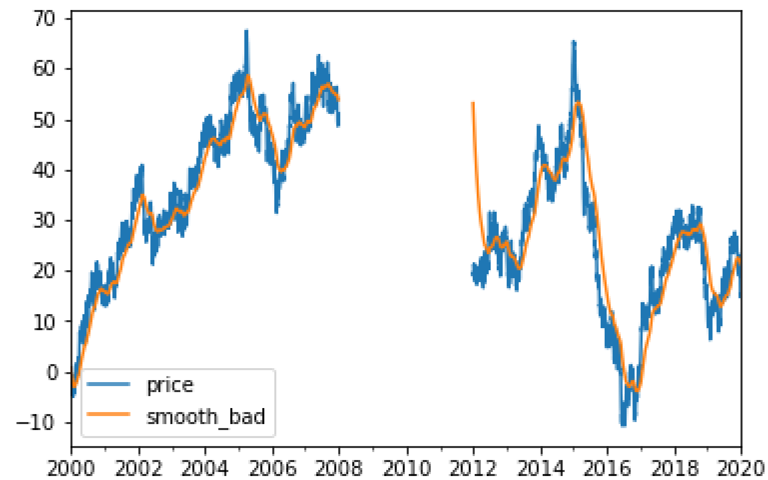
[1]:
```
from pyg import *; import numpy as np; import pandas as pd
rtn = pd.Series(np.random.normal(0.01,1,5218), drange(2000,2020, '1b'))
price = cumsum(rtn); price.name = 'price'
smooth = ewma(price, 50); smooth.name = 'smooth'
pd.concat([price, smooth], axis = 1).plot()
```

[1]:
```
<AxesSubplot:>
```



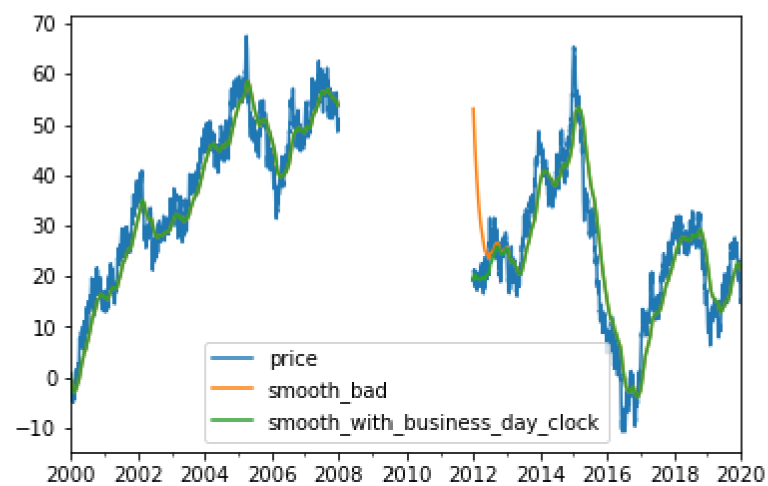
[2]:
```
## now suppose somewhow we lost 4 years of data...
bad = price.copy()
bad[dt(2008):dt(2012)] = np.nan
smooth_bad = ewma(bad, 50); smooth_bad.name = 'smooth_bad'
pd.concat([bad, smooth_bad], axis = 1).plot()
```

[2]:
```
<AxesSubplot:>
```



[8]:
```
smooth_with_time = ewma(bad, 50, time = 'b')
smooth_with_time.name = 'smooth_with_business_day_clock'
pd.concat([bad, smooth_bad, smooth_with_time], axis = 1).plot()
```

[8]: <AxesSubplot:>



What happened here? How can smooth with clock track better? The answer is that if you provide a clock, ewma can recognise that 4 years have passed. The old data is irrelevant, it forgets the old position and start with most of the weight on the more recent observations

13.1 What happens if the clock does not move at all?

Suppose we now choose to calculate daily ewma but we are doing this with intraday data. If the number of data points per day is constant and known, then this can be done with ease. Using time parameter, we can do this even for an irregularly spaced timeseries. We first just create some fake data:

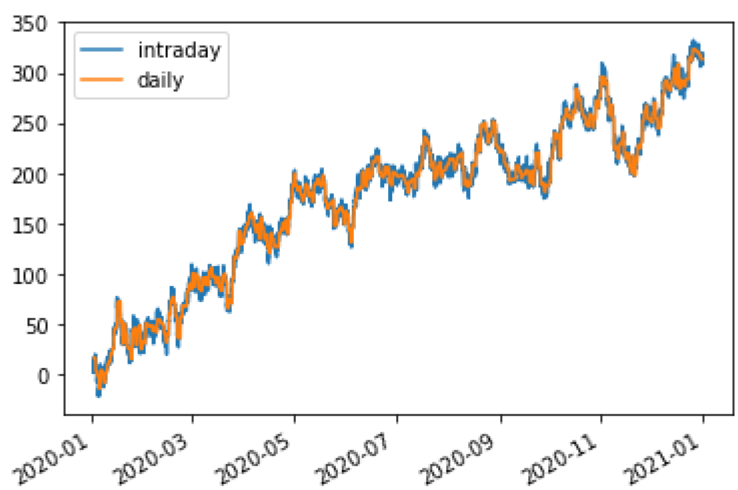
[4]:
```
import datetime
bar = datetime.timedelta(minutes = 5)
all_bars = [t for t in drange(2020, 2021, bar)]
```

(continues on next page)

(continued from previous page)

```
ts = pd.Series(np.random.normal(0.01/24, 1, len(all_bars)), all_bars)
price = cumsum(ts); price.name = 'intraday'

bars = np.array([t for t in drange(2020, 2021, bar) if t.hour>=8 and t.hour<=17]) ##␣
↪trading hours
irregular = bars[np.random.normal(0,1,len(bars))>0]

ts = pd.Series(np.random.normal(0.01/24, 1, len(bars)), bars)
price = cumsum(ts); price.name = 'intraday'
price = price[irregular] ## remove half the bars randomly

days = drange(2020,2021,1)

daily = price.reindex(days, method = 'ffill'); daily.name = 'daily'
pd.concat([price, daily], axis = 1).ffill().plot()
```

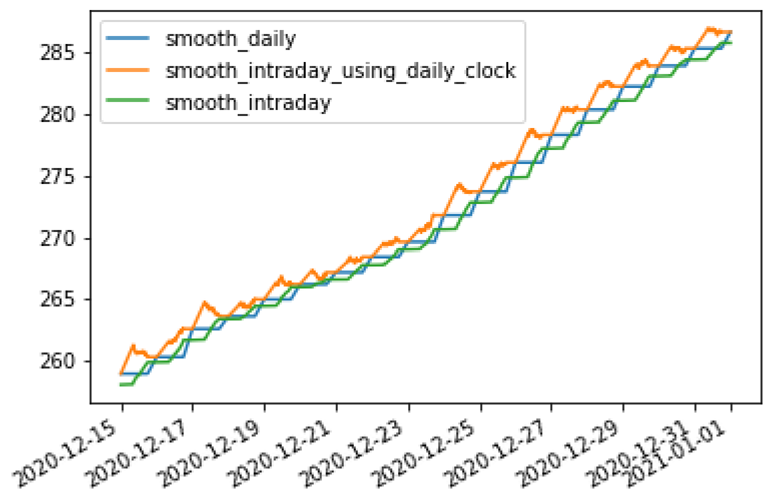
[4]: <AxesSubplot:>



- By setting clock to **d**aily we tell ewma that all the hourly data in the same day are 'on same clock'. And it will use historic end-of-day prices while updating today the last point until the cloc moves to tomorrow's reading
- If we set the clock to **f**raction, it will update continuously throuout the day

[5]:
```
smooth_intra = ewma(price, 20, 'f'); smooth_intra.name = 'smooth_intraday' ## roughly␣
↪matching using irregular bars
smooth_intra_using_d = ewma(price, 20, time = 'd'); smooth_intra_using_d.name =
↪'smooth_intraday_using_daily_clock'
smooth_daily = ewma(daily, 20); smooth_daily.name = 'smooth_daily'
pd.concat([smooth_daily, smooth_intra_using_d, smooth_intra], axis = 1).
↪ffill()[dt(2020,12,15):].plot()
```

[5]: <AxesSubplot:>



- smooth_daily is calculated on daily basis and is constant within the day and experiences jumps on EOD
- time = 'd' option front-runs daily, but at the price of being more volatile intra-day. On end-of-day the two version aggree
- time = 'f' is a smoother version of daily. It is leading, but not by much

[7]:
```
pd.concat([smooth_intra_using_d.reindex(days, method = 'ffill'), smooth_daily], axis␣
↪= 1)
## on end-of-day we have an exact match between time = 'd' and daily smooth
```

[7]:
```
            smooth_intraday_using_daily_clock  smooth_daily
2020-01-01                                NaN           NaN
2020-01-02                          16.752182     16.752182
2020-01-03                           9.725194      9.725194
2020-01-04                           5.864732      5.864732
2020-01-05                           0.578778      0.578778
...                                       ...           ...
2020-12-28                         280.375383    280.375383
2020-12-29                         282.254090    282.254090
2020-12-30                         283.934041    283.934041
2020-12-31                         285.343263    285.343263
2021-01-01                         286.681044    286.681044

[367 rows x 2 columns]
```

13.2 What are valid time parameters?

- None: If None is provided, any (non-nan) observation is considered to be a clock ticking
- i: index of timeseries. The clock ticks also for nan observations. This is the default for pandas
- f: fraction of day
- b/d/w/m/q/y: business day/daily/weekly/monthly/quarterly or yearly
- Calendar: the business day as defined by the calendar provided
- For full control, you can provide a timeseries of non-decreasing times matching the original array

```
[ ]:
```

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

pyg.base._getitem, 50
pyg.base._inspect, 52
pyg.base._types, 47

INDEX

Symbols

__add__() (*in module pyg.base._dictattr.dictattr*), 4
__and__() (*in module pyg.base._dictattr.dictattr*), 4
__call__() (*in module pyg.base._dict.Dict*), 6
__call__() (*in module pyg.base._dictable.dictable*), 14
__sub__() (*in module pyg.base._dictattr.dictattr*), 4

A

add_() (*in module pyg.timeseries._index*), 115
address() (*pyg.mongo._reader.mongo_reader property*), 62
adjust() (*pyg.base._drange.Calendar method*), 32
argspec_add() (*in module pyg.base._inspect*), 52
argspec_defaults() (*in module pyg.base._inspect*), 52
argspec_required() (*in module pyg.base._inspect*), 52
argspec_update() (*in module pyg.base._inspect*), 53
as_list() (*in module pyg.base._as_list*), 43
as_time() (*in module pyg.base._drange*), 34
as_tuple() (*in module pyg.base._as_list*), 43

B

bfill() (*in module pyg.timeseries._rolling*), 78
bson2np() (*in module pyg.base._encode*), 28
bson2pd() (*in module pyg.base._encode*), 28

C

Calendar (*class in pyg.base._drange*), 31
calendar() (*in module pyg.base._drange*), 34
call_with_callargs() (*in module pyg.base._inspect*), 53
callattr() (*in module pyg.base._getitem*), 50
callitem() (*in module pyg.base._getitem*), 51
capitalize() (*in module pyg.base._txt*), 36
cell (*class in pyg.base._cell*), 22
cell_func() (*in module pyg.base._cell*), 24
cell_go() (*in module pyg.base._cell*), 23
cell_item() (*in module pyg.base._cell*), 24
clock() (*in module pyg.base._drange*), 34
clone() (*pyg.mongo._pk_reader.mongo_pk_reader method*), 66
clone() (*pyg.mongo._reader.mongo_reader method*), 63
Cmp() (*in module pyg.base._sort*), 45
cmp() (*in module pyg.base._sort*), 45
collection() (*pyg.mongo._reader.mongo_reader property*), 63
common_prefix() (*in module pyg.base._txt*), 38
concat() (*pyg.base._dictable.dictable class method*), 9
copy() (*pyg.base._cell.cell method*), 23
copy() (*pyg.base._dictattr.dictattr method*), 3
copy() (*pyg.base._ulist.ulist method*), 5
count() (*pyg.mongo._reader.mongo_reader method*), 63
create_index() (*pyg.mongo._pk_reader.mongo_pk_reader method*), 66
csv_encode() (*in module pyg.mongo._encoders*), 28
csv_write() (*in module pyg.mongo._encoders*), 67
cumprod() (*in module pyg.timeseries._expanding*), 89
cumprod_() (*in module pyg.timeseries._expanding*), 109
cumsum() (*in module pyg.timeseries._expanding*), 88
cumsum_() (*in module pyg.timeseries._expanding*), 109

D

date_range() (*in module pyg.base._drange*), 31
db_cell (*class in pyg.mongo._db_cell*), 68
db_load() (*in module pyg.mongo._db_cell*), 70
db_ref() (*in module pyg.mongo._db_cell*), 70
db_save() (*in module pyg.mongo._db_cell*), 69
decode() (*in module pyg.base._encode*), 26
dedup() (*pyg.mongo._pk_reader.mongo_pk_reader method*), 66
delete_many() (*pyg.mongo._cursor.mongo_cursor method*), 59
delete_one() (*pyg.mongo._cursor.mongo_cursor method*), 60
df_fillna() (*in module pyg.timeseries._index*), 111
df_index() (*in module pyg.timeseries._index*), 112
df_reindex() (*in module pyg.timeseries._index*), 113
Dict (*class in pyg.base._dict*), 5

dictable (*class in pyg.base._dictable*), 6
dictattr (*class in pyg.base._dictattr*), 3
diff() (*in module pyg.timeseries._rolling*), 72
diff_() (*in module pyg.timeseries._rolling*), 108
distinct() (*pyg.mongo._reader.mongo_reader method*), 63
div_() (*in module pyg.timeseries._index*), 115
do() (*pyg.base._dict.Dict method*), 6
do() (*pyg.base._dictable.dictable method*), 9
docs() (*pyg.mongo._pk_reader.mongo_pk_reader method*), 66
docs() (*pyg.mongo._reader.mongo_reader method*), 63
drange() (*in module pyg.base._drange*), 30
drop() (*pyg.mongo._cursor.mongo_cursor method*), 60
dt() (*in module pyg.base._dates*), 28
dt_bump() (*in module pyg.base._dates*), 30
dt_bump() (*pyg.base._drange.Calendar method*), 32

E

encode() (*in module pyg.base._encode*), 25
eq() (*in module pyg.base._eq*), 46
ewma() (*in module pyg.timeseries._ewm*), 99
ewma_() (*in module pyg.timeseries._ewm*), 110
ewmcor() (*in module pyg.timeseries._ewm*), 106
ewmcor_() (*in module pyg.timeseries._ewm*), 110
ewmrms() (*in module pyg.timeseries._ewm*), 101
ewmrms_() (*in module pyg.timeseries._ewm*), 110
ewmskew() (*in module pyg.timeseries._ewm*), 107
ewmskew_() (*in module pyg.timeseries._ewm*), 111
ewmstd() (*in module pyg.timeseries._ewm*), 103
ewmstd_() (*in module pyg.timeseries._ewm*), 110
ewmvar() (*in module pyg.timeseries._ewm*), 104
ewmvar_() (*in module pyg.timeseries._ewm*), 110
exc() (*pyg.base._dictable.dictable method*), 10
exc() (*pyg.mongo._reader.mongo_reader method*), 63
expanding_max() (*in module pyg.timeseries._max*), 86
expanding_max_() (*in module pyg.timeseries._max*), 109
expanding_mean() (*in module pyg.timeseries._expanding*), 79
expanding_mean_() (*in module pyg.timeseries._expanding*), 109
expanding_median() (*in module pyg.timeseries._median*), 87
expanding_min() (*in module pyg.timeseries._min*), 85
expanding_min_() (*in module pyg.timeseries._min*), 109
expanding_rank() (*in module pyg.timeseries._rank*), 88
expanding_rms() (*in module pyg.timeseries._expanding*), 80
expanding_rms_() (*in module pyg.timeseries._expanding*), 109
expanding_skew() (*in module pyg.timeseries._expanding*), 84
expanding_skew_() (*in module pyg.timeseries._expanding*), 109
expanding_std() (*in module pyg.timeseries._expanding*), 81
expanding_std_() (*in module pyg.timeseries._expanding*), 109
expanding_sum() (*in module pyg.timeseries._expanding*), 83
expanding_sum_() (*in module pyg.timeseries._expanding*), 109

F

ffill() (*in module pyg.timeseries._rolling*), 78
ffill_() (*in module pyg.timeseries._rolling*), 109
find() (*pyg.mongo._reader.mongo_reader method*), 64
find_one() (*pyg.mongo._reader.mongo_reader method*), 64
first() (*in module pyg.base._as_list*), 44
fnna() (*in module pyg.timeseries._rolling*), 77

G

get() (*pyg.base._dictable.dictable method*), 10
get_logger() (*in module pyg.base._logger*), 50
getargs() (*in module pyg.base._inspect*), 53
getargspec() (*in module pyg.base._inspect*), 53
getcallargs() (*in module pyg.base._inspect*), 54
getitem() (*in module pyg.base._getitem*), 51
groupby() (*pyg.base._dictable.dictable method*), 10

I

in_() (*in module pyg.base._eq*), 47
inc() (*pyg.base._dictable.dictable method*), 11
inc() (*pyg.mongo._reader.mongo_reader method*), 64
insert_many() (*pyg.mongo._cursor.mongo_cursor method*), 60
insert_one() (*pyg.mongo._cursor.mongo_cursor method*), 61
is_arr() (*in module pyg.base._types*), 47
is_bool() (*in module pyg.base._types*), 47
is_date() (*in module pyg.base._types*), 47
is_df() (*in module pyg.base._types*), 47
is_dict() (*in module pyg.base._types*), 48
is_float() (*in module pyg.base._types*), 48
is_int() (*in module pyg.base._types*), 48
is_iterable() (*in module pyg.base._types*), 48
is_len() (*in module pyg.base._types*), 48
is_list() (*in module pyg.base._types*), 48
is_nan() (*in module pyg.base._types*), 48
is_none() (*in module pyg.base._types*), 48
is_num() (*in module pyg.base._types*), 48

is_pd() (in module pyg.base._types), 48
is_series() (in module pyg.base._types), 48
is_str() (in module pyg.base._types), 48
is_trading() (pyg.base._drange.Calendar method), 32
is_ts() (in module pyg.base._types), 48
is_tuple() (in module pyg.base._types), 48
items_to_tree() (in module pyg.base._dict), 39

J

join() (in module pyg.base._perdictable), 16
join() (pyg.base._dictable.dictable method), 11

K

keys() (pyg.base._dictattr.dictattr method), 3
kwargs2args() (in module pyg.base._inspect), 54
kwargs_support() (in module pyg.base._decorators), 22

L

last() (in module pyg.base._as_list), 44
load() (pyg.mongo._db_cell.db_cell method), 68
loop() (in module pyg.base._loop), 22
loops (class in pyg.base._loop), 21
lower() (in module pyg.base._txt), 35

M

mkdir() (in module pyg.base._file), 38
module
    pyg.base._getitem, 50
    pyg.base._inspect, 52
    pyg.base._types, 47
mongo_cursor (class in pyg.mongo._cursor), 58
mongo_pk_reader (class in pyg.mongo._pk_reader), 66
mongo_reader (class in pyg.mongo._reader), 62
mul_() (in module pyg.timeseries._index), 115

N

na2v() (in module pyg.timeseries._rolling), 78
named_dict() (in module pyg.base._named_dict), 18
nan2none() (in module pyg.base._types), 48
nona() (in module pyg.timeseries._ts), 79
np2bson() (in module pyg.base._encode), 28

P

parquet_encode() (in module pyg.mongo._encoders), 27
parquet_write() (in module pyg.mongo._encoders), 67
pd2bson() (in module pyg.base._encode), 28
pd_read_parquet() (in module pyg.base._parquet), 27
pd_to_parquet() (in module pyg.base._parquet), 26
perdictable() (in module pyg.base._perdictable), 15
periodic_cell (class in pyg.mongo._periodic_cell), 69
pivot() (pyg.base._dictable.dictable method), 11
pow_() (in module pyg.timeseries._index), 115
presync() (in module pyg.timeseries._index), 113
project() (pyg.mongo._reader.mongo_reader method), 65
projection() (pyg.mongo._reader.mongo_reader property), 65
proper() (in module pyg.base._txt), 36
pyg.base._getitem
    module, 50
pyg.base._inspect
    module, 52
pyg.base._types
    module, 47

Q

Q (class in pyg.mongo._q), 57

R

ratio() (in module pyg.timeseries._rolling), 73
ratio_() (in module pyg.timeseries._rolling), 108
raw() (pyg.mongo._cursor.mongo_cursor property), 61
raw() (pyg.mongo._reader.mongo_reader property), 65
read() (pyg.mongo._reader.mongo_reader method), 65
read_csv() (in module pyg.base._file), 38
reducer() (in module pyg.base._reducer), 49
reducing (class in pyg.base._reducer), 50
relabel() (pyg.base._dictattr.dictattr method), 4
rename() (pyg.base._dictattr.dictattr method), 4
replace() (in module pyg.base._txt), 37
rolling_max() (in module pyg.timeseries._max), 96
rolling_max_() (in module pyg.timeseries._max), 110
rolling_mean() (in module pyg.timeseries._rolling), 90
rolling_mean_() (in module pyg.timeseries._rolling), 110
rolling_median() (in module pyg.timeseries._median), 97
rolling_median_() (in module pyg.timeseries._median), 110
rolling_min() (in module pyg.timeseries._min), 95
rolling_min_() (in module pyg.timeseries._min), 110
rolling_quantile() (in module pyg.timeseries._stride), 98
rolling_quantile_() (in module pyg.timeseries._stride), 110
rolling_rank() (in module pyg.timeseries._rank), 99

rolling_rank_() (*in module pyg.timeseries._rank*), 110
rolling_rms() (*in module pyg.timeseries._rolling*), 91
rolling_rms_() (*in module pyg.timeseries._rolling*), 110
rolling_skew() (*in module pyg.timeseries._rolling*), 94
rolling_skew_() (*in module pyg.timeseries._rolling*), 110
rolling_std() (*in module pyg.timeseries._rolling*), 92
rolling_std_() (*in module pyg.timeseries._rolling*), 110
rolling_sum() (*in module pyg.timeseries._rolling*), 93
rolling_sum_() (*in module pyg.timeseries._rolling*), 110

S

set() (*pyg.mongo._cursor.mongo_cursor method*), 61
shift() (*in module pyg.timeseries._rolling*), 72
shift_() (*in module pyg.timeseries._rolling*), 108
sort() (*in module pyg.base._sort*), 46
sort() (*pyg.base._dictable.dictable method*), 12
sort() (*pyg.mongo._reader.mongo_reader method*), 65
spec() (*pyg.mongo._reader.mongo_reader property*), 65
specify() (*pyg.mongo._reader.mongo_reader method*), 66
split() (*in module pyg.base._txt*), 37
strip() (*in module pyg.base._txt*), 36
sub_() (*in module pyg.timeseries._index*), 115

T

timer (*class in pyg.base._decorators*), 20
trade_date() (*pyg.base._drange.Calendar method*), 32
tree_repr() (*in module pyg.base._tree_repr*), 42
tree_to_items() (*in module pyg.base._dict*), 38
tree_to_table() (*in module pyg.base._tree*), 41
tree_update() (*in module pyg.base._dict*), 40
try_back() (*in module pyg.base._decorators*), 21
try_value() (*in module pyg.base._decorators*), 21
ts_count() (*in module pyg.timeseries._ts*), 73
ts_count_() (*in module pyg.timeseries._ts*), 108
ts_max() (*in module pyg.timeseries._ts*), 77
ts_max_() (*in module pyg.timeseries._ts*), 109
ts_mean() (*in module pyg.timeseries._ts*), 74
ts_mean_() (*in module pyg.timeseries._ts*), 108
ts_median() (*in module pyg.timeseries._ts*), 77
ts_rms() (*in module pyg.timeseries._ts*), 75
ts_rms_() (*in module pyg.timeseries._ts*), 109
ts_skew() (*in module pyg.timeseries._ts*), 76
ts_skew_() (*in module pyg.timeseries._ts*), 109
ts_std() (*in module pyg.timeseries._ts*), 76
ts_std_() (*in module pyg.timeseries._ts*), 109
ts_sum() (*in module pyg.timeseries._ts*), 74
ts_sum_() (*in module pyg.timeseries._ts*), 108

U

ulist (*class in pyg.base._ulist*), 5
ungroup() (*pyg.base._dictable.dictable method*), 12
unique() (*in module pyg.base._as_list*), 44
unlist() (*pyg.base._dictable.dictable method*), 13
unpivot() (*pyg.base._dictable.dictable method*), 13
update() (*pyg.base._dictable.dictable method*), 13
update_many() (*pyg.mongo._cursor.mongo_cursor method*), 62
update_one() (*pyg.mongo._cursor.mongo_cursor method*), 62
upper() (*in module pyg.base._txt*), 35

V

v2na() (*in module pyg.timeseries._rolling*), 77
values() (*pyg.base._dictattr.dictattr method*), 4

W

wrapper (*class in pyg.base._decorators*), 19

X

xor() (*pyg.base._dictable.dictable method*), 13
xyz() (*pyg.base._dictable.dictable method*), 14

Y

ymd() (*in module pyg.base._dates*), 30

Z

zipper() (*in module pyg.base._zip*), 48