Pyg documentation

version

Yoav Git

October 25, 2021

Contents

Welcome to pyg!

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.

pyg.base

extensions to dict

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**)**

easy relabel/rename of keys

Parameters:

***args :** *str or callable*

- a string ending/starting with _ will trigger a prefix/suffix to all keys
- callable function will be applied to the keys to update them

****relabels :** *strings*

individual relabeling of keys

Returns:

dictattr

new dict with renamed keys.

Example: suffix/prefix

```
>>> from pyg import *
>>> d = dictattr(a = 1, b = 2, c = 3)
>>> assert d.relabel('x_') == dictattr(x_a = 1, x_b = 2, x_c = 3) # prefixing
>>> assert d.relabel('_x') == dictattr(a_x = 1, b_x = 2, c_x = 3) # suffixing
```

Example: callable

```
>>> assert d.rename(upper) == dictattr(A = 1, B = 2, C = 3)
```

Example: individual relabelling

```
>>> assert d.rename(a = 'A') == dictattr(A = 1, b = 2, c = 3)
>>> assert d.rename(['A', 'B', 'C']) == d.relabel(upper)
```

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

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.

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.

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]
```

Example: construction using pandas.DataFrame

```
>>> original = dictable(name = ['alan', 'barbara'], surname = ['atkins', 'brown'], age = [
>>> 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 =
```

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 br
```

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, 'countr
>>>                               {'name': 'barbara', 'surname': 'brown', 'age': 29, 'coun
```

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 + surnam
>>> assert sum([row for row in d], dictable()) == d
```

Example: adding records

```
>>> d = dictable(name = ['alan', 'barbara'], surname = ['atkins', 'brown'], age = [39, 29]
>>> d = d + {'name': 'charlie', 'surname': 'chocolate', 'age': 49} # can add a record dire
>>> assert d[-1] == {'name': 'charlie', 'surname': 'chocolate', 'age': 49, 'country': None
>>> d += dictable(name = ['dana', 'ender'], surname = ['deutch', 'esterhase'], age = [10, 
>>> 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]
```

```
>>> ### 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]
>>> 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]
>>> del d.country # in place
>>> del d['age'] # in place
>>> assert (d - 'name')[0] ==  {'surname': 'atkins'} and d[0] == {'name': 'alan', 'surname
```

Example: row selection, inc/exc

```
>>> d = dictable(name = ['alan', 'barbara'], surname = ['atkins', 'brown'], age = [39, 29]
>>> assert len(d.exc(name = 'alan')) == 1
>>> assert len(d.exc(lambda age: age<30)) == 1 # can filter on *functions* of members, not
>>> 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', 'cohe
>>> assert d.do(proper) == dictable(name = ['Adam', 'Barbara', 'Chris'], surname = ['Atki
```

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 regex

```
>>> import re
>>> d = dictable(text = ['once', 'upon', 'a', 'time', 'in', 'the', 'west', 1, 2, 3])
>>> assert d.inc(text = re.compile('o')) == dictable(text = ['once', 'upon'])
>>> assert d.exc(text = re.compile('e')) == dictable(text = ['upon', 'a', 'in', 1, 2, 3]
```

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]
>>> 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 me
>>> 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]
```

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.

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 = No
```

# some parameters are constant, some are tables…

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

# multiple tables… some unkeyed

```
>>> assert p(a = dictable(a = [1,2]), b = dictable(key = ['a','b','c'], b = [1,2,3])) == d
```

# 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
>>> 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 recalculat
```

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

```
>>> ## 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 = [
```

```
>>> ## 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 =
```

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 = di
```

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 = [
>>> assert join(dict(x = x, y = y), on = 'a', defaults = dict(y = 0)) == dictable(a = [1,2
>>> assert join(dict(x = x, y = y), on = 'a', defaults = dict(x = None, y = 0)) == dictabl
```

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
```

Example: Adding defaults

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

Example: types checking

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

Example: casting

```
>>> Customer = named_dict('Customer', ['name', 'date', 'balance'], defaults = dict(balance
>>> 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

decorators

wrapper

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

A base class for all decorators. It is similar to functools.wraps but better. See below why wrapt cannot be used… 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
```

```
>>> @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
```

Example: No recursion

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
```

Example: wrapper vs wrapt

wrapt (wrapt.readthedocs.io) is an awesome wrapping tool. If you have static library functions, none is better. The problem we face is that wrapt is too good in pretending the wrapped up object is the same as original function:

```
>>> import wrapt
>>> def add_value(value):
>>>     @wrapt.decorator
>>>     def wrapper(wrapped, instance, args, kwargs):
>>>         return wrapped(*args, **kwargs) + value
>>>     return wrapper
```

```
>>> def f(x,y):
>>>     return x*y
```

```
>>> add_three = add_value(value = 3)(f)
>>> add_four = add_value(value = 4)(f)
>>> assert add_four(3,4) == 16 and add_three(3,4) == 15
```

```
>>> ## but here is the problem:
>>> assert encode(add_three) == encode(add_four) == encode(f)
```

So if we ever encode the function and send it across json/Mongo, the wrapping is lost and the user when she receives it cannot use it

```
>>> class add_value(wrapper):
>>>     def wrapped(self, *args, **kwargs):
>>>         return self.function(*args, **kwargs) + self.value
```

```
>>> add_three = add_value(value = 3)(f)
>>> add_four = add_value(value = 4)(f)
>>> encode(add_three)
>>> {'value': 3, 'function': '{"py/function": "__main__.f"}', '_obj': '{"py/type": "__main
>>> encode(add_three)
>>> {'value': 4, 'function': '{"py/function": "__main__.f"}', '_obj': '{"py/type": "__main
```

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)
```

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'
```

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
```

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]))
```

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

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
```

graphs & cells

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

go (go=1**,** mode=0**,** **kwargs**)**

calculates the cell (if needed). By default, will then run cell.save() to save the cell. If you don't want to save the output (perhaps you want to check it first), use cell._go()

Parameters:

go : *int, optional*

a parameter that forces calculation. The default is 0. go = 0: calculate cell only if cell.run() is True go = 1: calculate THIS cell regardless. calculate the parents only if their cell.run() is True go = 2: calculate THIS cell and PARENTS cell regardless, calculate grandparents if cell.run() is True etc. go = -1: calculate the entire tree again.

****kwargs :** *parameters*

You can actually allocate the variables to the function at runtime

Note that by default, cell.go() will default to go = 1 and force a calculation on cell while cell() is lazy and will default to assuming go = 0

Returns:

cell

the cell, calculated

Example: different values of go

```
>>> from pyg import *
>>> f = lambda x=None,y=None: max([dt(x), dt(y)])
>>> a = cell(f)()
>>> b = cell(f, x = a)()
>>> c = cell(f, x = b)()
>>> d = cell(f, x = c)()
```

```
>>> e = d.go()
>>> e0 = d.go(0)
>>> e1 = d.go(1)
>>> e2 = d.go(2)
>>> e_1 = d.go(-1)
```

```
>>> assert not d.run() and e.data == d.data
>>> assert e0.data == d.data
>>> assert e1.data > d.data and e1.x.data == d.x.data
>>> assert e2.data > d.data and e2.x.data > d.x.data and e2.x.x.data == d.x.x.data
>>> assert e_1.data > d.data and e_1.x.data > d.x.data and e_1.x.x.data > d.x.x.data
```

Example: adding parameters on the run

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

`load` **(**mode=0**)**

Loads the cell from the database based on primary keys of cell perhaps. Not implemented for simple cell. see db_cell

Returns:

cell

self, updated with values from database.

`register` **(**inputs=None**)**

registers the cell so that it calculates when the inputs of the cells are calculated

***inputs :** *strs*

list of inputs

cell

self.

`run` **()**

checks if the cell needs calculation. This depends on the nature of the cell. By default (for cell and db_cell), if the cell is already calculated so that cell._output exists, then returns False. otherwise True

bool

run cell?

Example:

```
>>> c = cell(lambda x: x+1, x = 1)
>>> assert c.run()
>>> c = c()
>>> assert c.data == 2 and not c.run()
```

`save` **()**

Saves the cell for persistency. Not implemented for simple cell. see db_cell

Returns:

cell

self, saved.

cell_go

`pyg.base._cell.`**`cell_go`** (value, go=0, mode=0)

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)
```

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*

cell

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
```

cell_func

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

cell_func is a wrapper 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.

In order to present the itemized value in the cell, inputs for the function that are cells will:

1. be loaded (from the persistency layer)
2. called and calculated
3. itemized: i.e. cell_item(input)

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 kwargs['a'].data == 3 ** 2
>>> assert kwargs['b'].data == 2 ** 3
```

Parameters:

function : *callable*

The function to be wrapped

relabels : *dict or None*

Allows a redirect of variable names. For example:

```
>>> from pyg import *
>>> a = cell(a = 1, b = 2, c = 3)
>>> f = cell_func(lambda x: x+1, relabels = dict(x = 'c')) ## please use key 'c' to grab x valu
>>> assert f(a)[0] == 4
```

unloaded: list or str

By defaults, if a cell is in the inputs, it will be loaded (cell.load()) prior to being presented to the function If an arg is in unloaded, it will not be loaded

uncalled: list or str

By defaults, if a cell is in the inputs, it will be called (cell.call()) prior to being presented to the function If an arg is in uncalled, it will not be called

unitemized: list or str

, if a cell is in the inputs, once run, we itemize and grab its data If an arg is in unitemized, it will be presented 'as is' :Example:

```
n pyg import *
 cell(passthru, data = 'this is the value presented')
ert cell_func(lambda x: len(x))(x)[0] == len('this is the value presented') ## function run on item
ert cell_func(lambda x: len(x), unitemized = 'x')(x)[0] == len(x)         ## function run on x as-
```

cell_clear

`pyg.base._cell.`**`cell_clear`** (value)

cell_clear clears a cell of its output so that it contains only the essentil stuff to do its calculations. This will be used when we save the cell or we want to recalculate it.

Example:

```
>>> from pyg import *
>>> a = cell(add_, a = 1, b = 2)
>>> b = cell(add_, a = 2, b = 3)
>>> c = cell(add_, a = a, b = b)()
>>> assert c.data == 8
>>> assert c.a.data == 3
```

```
>>> bare = cell_clear(c)
>>> assert 'data' not in bare and 'data' not in bare.a
>>> assert bare() == c
```

Parameters:

value: obj

cell (or list/dict of) to be cleared of output

encode and decode/save and load

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(
>>> value = dictable(a=[1,2,3], b = 4)
>>> assert encode(value) == {'a': [1, 2, 3], 'b': [4, 4, 4], '_obj': '{"py/type": "pyg.bas
>>> assert decode(encode(value)) == value
>>> assert encode(np.array([1,2])) ==  {'data': bytes,
>>>                                     'shape': (2,),
>>>                                     'dtype': '{"py/reduce": [{"py/type": "numpy.dtype"
>>>                                     '_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

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...
```

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)
```

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)
```

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(
>>> 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)
```

csv_encode

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

encodes a single DataFrame or a document containing dataframes into a an abject 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])],
>>> 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)
```

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.

dates and calendar

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

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()

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

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)))
```

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 week
```

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)
```

date_range

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

Calendar

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

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

```
>>> 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), d
```

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

add (date**,** days**,** adj=None**)**

adjustes the start date to a business day. Then add business days on top. add will initiate self._populate unless we are bumping date by exactly one day

date : *datetime*

start date.

days : *int*

days to bump.

adj : *str, optional*

adjusting method for date if it isn't a business day to start with.

datetime

bumped date.

`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**,** day_start=0**,** day_end=235959**)**
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**,** day_start=0**,** day_end=235959**)**
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
>>> assert uk.trade_date(dt(2021,2,9,5), 'p') == dt(2021, 2, 8)  # Tuesday morning back i
>>> assert uk.trade_date(dt(2021,2,7,5), 'f') == dt(2021, 2, 8)  # Sunday rolls into Mond
>>> assert uk.trade_date(dt(2021,2,7,5), 'p') == dt(2021, 2, 5)  # Sunday rolls back to F
>>> assert uk.trade_date(date = dt(2021,2,9,23), adj = 'f') == dt(2021, 2, 10)  # Tuesday
>>> assert uk.trade_date(date = dt(2021,2,9,23), adj = 'p') == dt(2021, 2, 9)  # Tuesday
>>> assert uk.trade_date(date = dt(2021,2,7,23), adj = 'f') == dt(2021, 2, 8)  # Sunday r
>>> assert uk.trade_date(date = dt(2021,2,7,23), adj = 'p') == dt(2021, 2, 5)  # Sunday r
```

```
>>> assert uk.trade_date(date = dt(2021,2,9,12), adj = 'f') == dt(2021, 2, 9)  # Tuesday
>>> assert uk.trade_date(date = dt(2021,2,9,12), adj = 'p') == dt(2021, 2, 9)  # 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 ses
>>> assert au.trade_date(dt(2021,2,9,5), 'p') == dt(2021, 2, 9)  # Tuesday morning in ses
>>> assert au.trade_date(dt(2021,2,7,5), 'f') == dt(2021, 2, 8)  # Sunday rolls into Mond
>>> assert au.trade_date(dt(2021,2,7,5), 'p') == dt(2021, 2, 5)  # Sunday rolls back to F
```

```
>>> assert au.trade_date(date = dt(2021,2,9,23), adj = 'f') == dt(2021, 2, 10)  # Tuesday
>>> assert au.trade_date(date = dt(2021,2,9,23), adj = 'p') == dt(2021, 2, 10)  # Already
>>> assert au.trade_date(date = dt(2021,2,7,23), adj = 'f') == dt(2021, 2, 8)  # Sunday r
>>> assert au.trade_date(date = dt(2021,2,7,23), adj = 'p') == dt(2021, 2, 8)  # Already
>>> assert au.trade_date(date = dt(2021,2,5,23), adj = 'f') == dt(2021, 2, 8)  # Friday a
```

```
>>> assert au.trade_date(date = dt(2021,2,9,14), adj = 'f') == dt(2021, 2, 10)  # Tuesday
>>> assert au.trade_date(date = dt(2021,2,9,14), adj = 'p') == dt(2021, 2, 9)  # roll bac
```

calendar

`pyg.base._drange.`**`calendar`** (key=None, holidays=None, weekend=None, t0=None, t1=None)

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

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)
```

Parameters:

t : *str/int/datetime.time/datetime.datetime*

time of day

Returns:

datetime.time

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, 2609
>>> assert eq(clock(pd.Series(np.arange(10), drange(2000, '9b', '1b')), 'b'), np.arange(26
```

```
>>> 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))
```

text manipulation

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}
```

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}
```

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}
```

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']
```

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',
>>> assert strip(dict(a = ' whatever you say  ', b = 3.0)) ==  {'a': 'whatever you say', '
```

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', '
```

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 lot
>>> assert replace('this, sentence? has! too, many, punctuations!', list(',?!.')) == 'this
>>> assert replace(dict(a = 1, b = [' text within a list ', 'and within a dict']), ' ') ==
```

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]
```

files & directory

mkdir

`pyg.base._file.`**`mkdir`** (path)

makes a new directory if not exists. It works if path is a filename too.

read_csv

`pyg.base._file.`**`read_csv`** (path, errors='replace', **fmt)

A light-weight csv reader, no conversion is done, nor do we insist equal number of columns per row. - by default, encoding error (unicode characters) are replaced. - fmt parameters are parameters for the csv.reader object, see https://docs.python.org/3/library/csv.html

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) We deliberately avoid creating a tree class so that the functionality is available on ordinary tree-like structures.

tree_keys

`pyg.base._dict.`**`tree_keys`** (tree, types=None)
returns the keys (branches) of a tree as a list of of tuples

Example:

```
>>> tree = dict(a = 1, b = dict(c = 2, d = 3, e = dict(f = 4)))
>>> assert tree_keys(tree) == [('a',), ('b', 'c'), ('b', 'd'), ('b', 'e', 'f')]
```

Parameters:

tree : tree (dict of dicts) types : types of dicts, optional

tree_values

`pyg.base._dict.`**`tree_values`** (tree, types=None)
returns the values (leaf) of a tree (a collection of tuples)

Example:

```
>>> tree = dict(a = 1, b = dict(c = 2, d = 3, e = dict(f = 4)))
>>> assert tree_values(tree) == [1,2,3,4]
```

Parameters:

tree : tree (dict of dicts) types : types of dicts, optional

tree_items

`pyg.base._dict.`**`tree_items`** (tree, types=None)
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 = '
                      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 = 
                        physics = dict(name = 'richard', surname = 'feyman', grade = 4)
                        ))
```

```
>>> items = tree_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
```

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_items. The default is (dict, Dict, dictattr).

Returns:

tree

updated tree.

tree_setitem

`pyg.base._dict.`**`tree_setitem`** (tree, key, value, ignore=None, types=None)
sets an item of a tree

Parameters:

tree : tree (dicts of dict) key : a dot-separated string or a tuple of values

the branch to hang value on

value : *object*

the leaf at the end of the branch

ignore : *None or list, optional*

what values of leaf will be ignored and not overwrite existing data. The default is None.

types : *types, optional*

As we go down the tree, when do we stop and say: what we have is a leaf already?

Example:

```
>>> tree = dict()
>>> tree_setitem(tree, 'a', 1)
>>> assert tree == dict(a = 1)
>>> tree_setitem(tree, 'b.c', 2)
>>> assert tree == {'a': 1, 'b': {'c': 2}}
>>> tree_setitem(tree, ('b','c','d'), 2)
>>> tree_setitem(tree, ('b','c','e'), 3)
>>> assert tree == {'a': 1, 'b': {'c': {'d': 2, 'e': 3}}}
```

Example: types

```
>>> from pyg import *
>>> tree = dict(mycell = cell(lambda a, b: a * b, b = 2, a = cell(lambda x: x**2, x = cell
>>> # We are missing y....
>>> tree_setitem(tree, 'mycell.a.x.y', 3, types = (dict,cell)) ## drill into cell
>>> assert tree['mycell'].a.x.y == 3
>>> tree_setitem(tree, 'mycell.a.x.y', 1) ## stop when you hit cell
>>> assert tree['mycell'].a.x == dict(y = 1)
```

None.

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 = '
>>>                       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 = 
>>>                       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:

string

a tree-like string representation of a dict-of-dicts.

items_to_tree

`pyg.base._dict.`**`items_to_tree`** (items, tree=None, raise_if_duplicate=True, ignore=None, types=None)

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

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 = '
                           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', grad
                             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

list functions

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.

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.

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
```

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
```

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])
```

Comparing and Sorting

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
```

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]
```

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]
```

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)
>>> 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
>>> 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.arr
>>> assert eq(dict(a = np.array([np.array([1,np.nan]),np.nan])) ,  dict(a = np.array([np.a
>>> assert eq(np.array([np.array([1,2]),dict(a = np.array([np.array([1,2]),2]))]), np.arra
```

```
>>> 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 =
>>> assert not eq(pd.DataFrame([1,np.nan], columns = ['a']), pd.DataFrame([1,np.nan], colu
```

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])
```

bits and pieces

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

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)]
>>> assert list(zipper([1,2,3], [4,5,6], None)) ==  [(1, 4, None), (2, 5, None), (3, 6, No
>>> assert list(zipper((1,2,3), np.array([4,5,6]), None)) ==  [(1, 4, None), (2, 5, 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

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, [])
```

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))

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

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 =
```

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')
```

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(**{'anno
                                                                   'args': ['a',
                                                                   'defaults': (
                                                                   'kwonlyargs':
                                                                   'kwonlydefaul
                                                                   'varargs': No
                                                                   '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,
```

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

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

`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.

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.

Query generator

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

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
>>> assert dict(q(a = re.compile('^hello'))) == {'a': {'regex': '^hello'}}     # a regex q
```

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

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

Tables in Mongo

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 allows you to transform the data before saving it in Mongo. You can create a function yourself or use built-in options:

- False: do nothing, save the document as is
- True/None: use pyg.base.encode to encode objects. This will transform numpy array/dataframes into bytes that can be stored
- '.csv': save dataframes into csv files and then save reference of these files to mongo
- '.parquet' save dataframes into .parquet and np.ndarray into .npy files.

For .csv and .parquet to work, you will need to specify WHERE the document is to be saved. This can be done either:

- the document has a 'root' key, specifying the root.
- you specify root by setting writer = 'c:/%name%surname.parquet'

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 us
```

Row 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
```

Example: root specification

```
>>> from pyg import *
>>> t = mongo_table('test', 'test', writer = 'c:/temp/%name/%surname.parquet')
>>> t.drop()
>>> doc = dict(name = 'adam', surname = 'smith', ts = pd.Series(np.arange(10)))
>>> t.insert_one(doc)
>>> assert eq(pd_read_parquet('c:/temp/adam/smith/ts.parquet'), doc['ts'])
>>> assert eq(t[0]['ts'], doc['ts'])
>>> doc = dict(name = 'beth', surname = 'brown', a = np.arange(10))
>>> t.insert_one(doc)
```

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]
>>> _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.

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(20
>>> table.insert_one(dict(name = 'barbara', surname = 'brown', age = 50, marriage = dt(20
>>> 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(20
>>> table.insert_one(dict(name = 'barbara', surname = 'brown', age = 50, marriage = dt(20
>>> 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(20
>>> table.insert_one(dict(name = 'barbara', surname = 'brown', age = 50, marriage = dt(20
>>> 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(20
>>> table.insert_one(dict(name = 'barbara', surname = 'brown', age = 50, marriage = dt(20
>>> 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

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

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:

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.

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

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

cells in Mongo

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

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()
```

go (go=1**,** mode=0**,** **kwargs**)**

calculates the cell (if needed). By default, will then run cell.save() to save the cell. If you don’t want to save the output (perhaps you want to check it first), use cell._go()

Parameters:

go : *int, optional*

a parameter that forces calculation. The default is 0. go = 0: calculate cell only if cell.run() is True go = 1: calculate THIS cell regardless. calculate the parents only if their cell.run() is True go = 2: calculate THIS cell and PARENTS cell regardless, calculate grandparents if cell.run() is True etc. go = -1: calculate the entire tree again.

****kwargs :** *parameters*

You can actually allocate the variables to the function at runtime

Note that by default, cell.go() will default to go = 1 and force a calculation on cell while cell() is lazy and will default to assuming go = 0

Returns:

cell

the cell, calculated

Example: different values of go

```
>>> from pyg import *
>>> f = lambda x=None,y=None: max([dt(x), dt(y)])
>>> a = cell(f)()
>>> b = cell(f, x = a)()
>>> c = cell(f, x = b)()
>>> d = cell(f, x = c)()
```

```
>>> e = d.go()
>>> e0 = d.go(0)
>>> e1 = d.go(1)
>>> e2 = d.go(2)
>>> e_1 = d.go(-1)
```

```
>>> assert not d.run() and e.data == d.data
>>> assert e0.data == d.data
>>> assert e1.data > d.data and e1.x.data == d.x.data
>>> assert e2.data > d.data and e2.x.data > d.x.data and e2.x.x.data == d.x.x.data
>>> assert e_1.data > d.data and e_1.x.data > d.x.data and e_1.x.x.data > d.x.x.data
```

Example: adding parameters on the run

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

load (mode=0**)**

loads a document from the database and updates various keys.

Persistency:

Since we want to avoid hitting the database, there is a singleton GRAPH, a dict, storing the cells by their address. Every time we load/save from/to Mongo, we also update GRAPH.
We use the GRAPH often so if you want to FORCE the cell to go to the database when loading, use this:

```
>>> cell.load(-1)
>>> cell.load(-1).load(0)  # clear GRAPH and load from db
>>> cell.load([0])     # same thing: clear GRAPH and then load if available
```

Merge of cached cell and calling cell:

Once we load from memory (either MongoDB or GRAPH), we tree_update the cached cell with the new values in the current cell. This means that it is next to impossible to actually *delete* keys. If you want to delete keys in a cell/cells in the database, you need to:

```
>>> del db.inc(filters)['key.subkey']
```

Parameters:

mode : *int , dataetime, optional*

if -1, then does not load and clears the GRAPH if 0, then will load from database if found. If not found, will return original document if 1, then will throw an exception if no document is found in the database if mode is a date, will return the version alive at that date The default is 0.
IF you enclose any of these in a list, then GRAPH is cleared prior to running and the database is called.

Returns:

document

save ()

Saves the cell for persistency. Not implemented for simple cell. see db_cell

Returns:

cell

self, saved.

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()
```

`run` **()**

checks if the cell needs calculation. This depends on the nature of the cell. By default (for cell and db_cell), if the cell is already calculated so that cell._output exists, then returns False. otherwise True

bool

run cell?

Example:

```
>>> c = cell(lambda x: x+1, x = 1)
>>> assert c.run()
>>> c = c()
>>> assert c.data == 2 and not c.run()
```

get_cell

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'
```

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

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/resampling. E.g.:**
    - 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 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.

simple functions

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.

Parameters:

a : *array/timeseries*

array/timeseries

n: int, optional, default = 1

window size

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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]))
```

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

data: None.
unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional
state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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 t
```

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:])
```

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

data: None.
unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional
state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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)))
```

ts_count

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

- supports numpy arrays
- skips nan
- supports state management

Example: pandas matching

```
>>> # 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
>>> 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:], state = old.state)
>>> assert new == ts_count(a)
```

ts_sum

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

- supports numpy arrays
- handles nan
- supports state management

Parameters:

a : *array, pd.Series, pd.DataFrame or list/dict of these*
timeseries

axis : *int, optional*
0/1/-1. The default is 0.

data: None
unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional
state parameters used to instantiate the internal calculations, based on history prior to ‘a’ provided.

Example: pandas matching

```
>>> # 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
>>> 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)
```

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

Parameters:

a : *array, pd.Series, pd.DataFrame or list/dict of these*

timeseries

axis : *int, optional*

0/1/-1. The default is 0.

data: None

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

Example: pandas matching

```
>>> # 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
>>> 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)
```

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

Parameters:

a : *array, pd.Series, pd.DataFrame or list/dict of these*

timeseries

axis : *int, optional*

0/1/-1. The default is 0.

data: None

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

- 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)
```

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)
```

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

Parameters:

a : *array, pd.Series, pd.DataFrame or list/dict of these*
    timeseries

axis : *int, optional*
    0/1/-1. The default is 0.

min_sample: float, optional
    This refers to the denominator when we calculate the skew. Over time, the deonimator converges to 1 but initially, it is small. Also, if there is a gap in the data, older datapoints weight may have decayed while there are not enough "new point". min_sample ensures that in both cases, if denominator<0.25)(default value) we return nan.

data: None
    unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional
    state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

Example: pandas matching

```
>>> # 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
>>> 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)
```

ts_min

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

ts_max

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

ts_median

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

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)
```

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

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

Parameters:

a : *array/timeseries*

array/timeseries

n: int, optional, default = 1

window size

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to ‘a’ provided.

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)
```

nona

`pyg.timeseries._ts.`**`nona`** (a, value=nan)
removes rows that are entirely nan (or a specific other value)

Parameters:

a : 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)
```

expanding window functions

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.

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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 = expan
>>> assert eq(expanding_mean([a,a**2]), [expanding_mean(a), expanding_mean(a**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.

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to ‘a’ provided.

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 = expandi
>>> assert eq(expanding_rms([a,a**2]), [expanding_rms(a), expanding_rms(a**2)])
```

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.

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to ‘a’ provided.

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 = expandi
>>> assert eq(expanding_std([a,a**2]), [expanding_std(a), expanding_std(a**2)])
```

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.

data: None

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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 = expandi
>>> assert eq(expanding_sum([a,a**2]), [expanding_sum(a), expanding_sum(a**2)])
```

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.

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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 = expan
>>> assert eq(expanding_skew([a,a**2]), [expanding_skew(a), expanding_skew(a**2)])
```

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.

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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 = expandi
>>> assert eq(expanding_min([a,a**2]), [expanding_min(a), expanding_min(a**2)])
```

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.

data: None.
unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional
state parameters used to instantiate the internal calculations, based on history prior to ‘a’ provided.

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 = expandi
>>> assert eq(expanding_max([a,a**2]), [expanding_max(a), expanding_max(a**2)])
```

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
>>>                  ...       ...
>>> 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 = e
>>> assert eq(expanding_median([a,a**2]), [expanding_median(a), expanding_median(a**2)])
```

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

Parameters:

a : *array, pd.Series, pd.DataFrame or list/dict of these*

timeseries

axis : *int, optional*

0/1/-1. The default is 0.

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)
```

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.

data: None
  unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional
  state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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 = expandi
>>> assert eq(expanding_sum([a,a**2]), [expanding_sum(a), expanding_sum(a**2)])
```

cumprod

`pyg.timeseries._expanding.`**`cumprod`** (a, axis=0, data=None, state=None)
  equivalent to pandas np.exp(np.log(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.

data: None
unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional
state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

Example: agreement with pandas

```
>>> from pyg import *; import pandas as pd; import numpy as np
>>> a = 1 + pd.Series(np.random.normal(0.001,0.05,10000), drange(-9999))
>>> panda = np.exp(np.log(a).expanding().sum()); ts = cumprod(a)
>>> assert abs(ts-panda).max() < 1e-10
```

Example: nan handling

Unlike pandas, timeseries does not forward fill the nans.

```
>>> a = 1 + pd.Series(np.random.normal(-0.01,0.05,100), drange(-99, 2020))
>>> a[a<0.975] = np.nan
>>> panda = np.exp(np.log(a).expanding().sum()); ts = cumprod(a)
```

```
>>> pd.concat([panda,ts], axis=1)
>>> 2019-09-24  1.037161  1.037161
>>> 2019-09-25  1.050378  1.050378
>>> 2019-09-26  1.158734  1.158734
>>> 2019-09-27  1.158734       NaN
>>> 2019-09-28  1.219402  1.219402
>>>               ...       ...
>>> 2019-12-28  4.032919  4.032919
>>> 2019-12-29  4.032919       NaN
>>> 2019-12-30  4.180120  4.180120
>>> 2019-12-31  4.180120       NaN
>>> 2020-01-01  4.244261  4.244261
```

Example: state management

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

```
>>> old = a.iloc[:50]
>>> new = a.iloc[50:]
>>> ts = cumprod(a)
>>> old_ts = cumprod_(old)
>>> new_ts = cumprod(new, **old_ts)
>>> assert eq(new_ts, ts.iloc[50:])
```

Example: dict/list inputs

```
>>> assert eq(cumprod(dict(x = a, y = a**2)), dict(x = cumprod(a), y = cumprod(a**2)))
>>> assert eq(cumprod([a,a**2]), [cumprod(a), cumprod(a**2)])
```

rolling window functions

rolling_mean

`pyg.timeseries._rolling.`**`rolling_mean`** (a, n, time=None, 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

time: a sequence of rising values of time

passage of time

axis : *int, optional*

0/1/-1. The default is 0.

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to ‘a’ provided.

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(pan
>>> #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 = rol
>>> assert eq(rolling_mean([a,a**2],10), [rolling_mean(a,10), rolling_mean(a**2,10)])
```

Example: passage of time

```
>>> a = np.array([1.,2.,3.,4.])
>>> time = np.array([0,0,1,1]) ## i.e. the first two observations are from day 0 and the n
>>> rolling_mean(a,2,time)
>>> array([nan, nan, 2.5, 3. ])
```

## The first two observations are from day 0 so cannot have a mean until we get a second point ## The mean is then calculated from last observation in day 0 (i.e. 2) and then 3. and then with 4. since these are again, from same day

rolling_rms

`pyg.timeseries._rolling.`**`rolling_rms`** (a, n, time=None, 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

time: a sequence of rising values of time
: passage of time

axis : *int, optional*
: 0/1/-1. The default is 0.

data: None.
: unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional
: state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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(pan
>>> #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 = rolli
>>> assert eq(rolling_rms([a,a**2],10), [rolling_rms(a,10), rolling_rms(a**2,10)])
```

Example: passage of time

```
>>> a = np.array([1.,2.,3.,4.])
>>> time = np.array([0,0,1,1]) ## i.e. the first two observations are from day 0 and the n
>>> (rolling_rms(a,2,time) ** 2) * 2
>>> array([nan, nan, 13., 20.])   == array([nan, nan, 4 + 9, 4 + 16])
```

## The first two observations are from day 0 so cannot have a mean until we get a second point ## The rms is then calculated from last observation in day 0 (i.e. 2) and then 3. and then with 4. since these are again, from same day

rolling_std

`pyg.timeseries._rolling.`**`rolling_std`** (a, n, time=None, 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

time: a sequence of rising values of time

passage of time

axis : *int, optional*

0/1/-1. The default is 0.

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to ‘a’ provided.

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(pan
>>> #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 = rolli
>>> assert eq(rolling_std([a,a**2],10), [rolling_std(a,10), rolling_std(a**2,10)])
```

Example: passage of time

```
>>> a = np.array([1.,2.,3.,4.])
>>> time = np.array([0,0,1,1]) ## i.e. the first two observations are from day 0 and the n
>>> 2*(rolling_std(a,2,time) ** 2)
>>> array([nan, nan, 1., 4.])
```

## The first two observations are from day 0 so cannot have a mean until we get a second point ## The sum is then calculated from last observation in day 0 (i.e. 2) and then 3. and then with 4. since these are again, from same day

rolling_sum

`pyg.timeseries._rolling.`**`rolling_sum`** (a, n, time=None, 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

time: a sequence of rising values of time
passage of time

axis : *int, optional*
0/1/-1. The default is 0.

data: None.
unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional
state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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(pan
>>> #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 = rolli
>>> assert eq(rolling_sum([a,a**2],10), [rolling_sum(a,10), rolling_sum(a**2,10)])
```

Example: passage of time

```
>>> a = np.array([1.,2.,3.,4.])
>>> time = np.array([0,0,1,1]) ## i.e. the first two observations are from day 0 and the n
>>> rolling_sum(a,2,time)
>>> array([nan, nan, 5., 6.])   == array([nan, nan, 2+3, 2+4])
```

## The first two observations are from day 0 so cannot have a mean until we get a second point ## The sum is then calculated from last observation in day 0 (i.e. 2) and then 3. and then with 4. since these are again, from same day

rolling_skew

`pyg.timeseries._rolling.`**`rolling_skew`** (a, n, bias=False, time=None, 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

time: a sequence of rising values of time

passage of time

bias:

affects the skew calculation definition, see scipy documentation for details.

axis : *int, optional*

0/1/-1. The default is 0.

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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(pan
>>> #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 = rol
>>> assert eq(rolling_skew([a,a**2],10), [rolling_skew(a,10), rolling_skew(a**2,10)])
```

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.

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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(pan
>>> #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 = rolli
>>> assert eq(rolling_min([a,a**2],10), [rolling_min(a,10), rolling_min(a**2,10)])
```

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.

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to ‘a’ provided.

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(pan
>>> #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 = rolli
>>> assert eq(rolling_max([a,a**2],10), [rolling_max(a,10), rolling_max(a**2,10)])
```

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.

data: None.
  unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional
  state parameters used to instantiate the internal calculations, based on history prior to ‘a’ provided.

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(pan
#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 =
>>> assert eq(rolling_median([a,a**2],10), [rolling_median(a,10), rolling_median(a**2,10)]
```

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).

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to ‘a’ provided.

Returns:

timeseries/array of quantile(s)

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

Parameters:

a : *array, pd.Series, pd.DataFrame or list/dict of these*

timeseries

n: int

window size

axis : *int, optional*

0/1/-1. The default is 0.

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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))
```

exponentially weighted moving functions

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 in time unit.

- 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)

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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
>>> 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.norma
>>> 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

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 in time unit.

- 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)

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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,
>>> 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.norma
>>> 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

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 in time unit.

- 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)

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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' pas
>>> assert abs(ts-df).max()<1e-10
>>> ts = ewmstd(a,10, time = 'i', bias = True); df = a.ewm(10).std(bias = True) # note: pa
>>> 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
>>> 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.norma
>>> 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

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 in time unit.

- 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)

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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:]
>>> 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
>>> 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

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 in time unit.

- 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)

data: None.

unused at the moment. Allow code such as func(live, **func_(history)) to work

state: dict, optional

state parameters used to instantiate the internal calculations, based on history prior to 'a' provided.

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' pas
>>> assert abs(ts-df).max()<1e-10
>>> ts = ewmvar(a,10, time = 'i', bias = True); df = a.ewm(10).var(bias = True) # note: pa
>>> 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.norma
>>> 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

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.

Parameters:

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 in time unit.

- 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, 'tim
>>> 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:])
```

ewmcorr

`pyg.timeseries._ewm.`**`ewmcorr`** (a, n, min_sample=0.25, bias=False, instate=None)

This calculates a full correlation matrix as a timeseries. The calculation is returned as an xarray.Dataset object, which is a multidimensional data structure.

Parameters:

a : *np.array or a pd.DataFrame*

timeseries to calculate correlation for

n : *int*

days for which rolling correlation is calculated.

min_sample : *float, optional*

Minimum observations needed before we calculate correlation. The default is 0.25.

bias : *bool, optional*

input to stdev calculations, the default is False.

instate : *dict, optional*

historical calculations so far.

correlation dataset

an xarray.Dataset unless there are only two timeserieses

Example: a pair of ts

```
>>> rtn = np.random.normal(0,1,10000)
>>> x0 = ewmxo(rtn, 10, 20, 30)[50:]
>>> x1 = ewmxo(rtn, 20, 40, 30)[50:]
>>> a = pd.DataFrame(np.array([x0,x1]).T, drange(-9949))
>>> res = ewmcorr(a, n)
>>> res
>>>     Out[130]:
>>>     1994-06-07         NaN
>>>     1994-06-08         NaN
>>>     1994-06-09         NaN
>>>     1994-06-10         NaN
>>>     1994-06-11         NaN
```

```
>>>     2021-08-29    0.890766
>>>     2021-08-30    0.886926
>>>     2021-08-31    0.883054
>>>     2021-09-01    0.879577
>>>     2021-09-02    0.875766
>>>     Length: 9950, dtype: float64
```

Example: a pair of ts

```
>>> rtn = np.random.normal(0,1,10000)
>>> x0 = ewmxo(rtn, 10, 20, 30)[50:]
>>> x1 = ewmxo(rtn, 20, 40, 30)[50:]
>>> x2 = ewmxo(rtn, 40, 80, 30)[50:]
>>> a = pd.DataFrame(np.array([x0,x1,x2]).T, drange(-9949), ['a','b','c'])
>>> ds = ewmcorr(a, n)
>>> ds
```

```
>>> <xarray.Dataset>
>>> Dimensions:  (index: 9950, x: 3, y: 3)
>>> Coordinates:
>>>   * index    (index) datetime64[ns] 1994-06-07 1994-06-08 ... 2021-09-02
>>>   * x        (x) <U1 'a' 'b' 'c'
>>>   * y        (y) <U1 'a' 'b' 'c'
>>> Data variables:
>>>     cor      (index, x, y) float64 1.0 nan nan nan ... 0.9402 0.7254 0.9402 1.0
```

To access individual correlations:

```
>>> a_vs_b = pd.Series(ds.loc[dict(x = 'a', y = 'b')].cor.values, ds.index.values)
```

To access all correlations to a:

```
>>> subset = ds.loc[dict(x = 'a')]
>>> a_vs_all = pd.DataFrame(subset.cor.values, ds.index.values, subset.y.values)
>>> a_vs_all
```

```
>>>                  a         b         c
>>> 1994-06-07  1.0       NaN       NaN
>>> 1994-06-08  1.0       NaN       NaN
>>> 1994-06-09  1.0       NaN       NaN
>>> 1994-06-10  1.0       NaN       NaN
>>> 1994-06-11  1.0       NaN       NaN
>>>         ...       ...       ...
>>> 2021-08-29  1.0  0.890766  0.753859
>>> 2021-08-30  1.0  0.886926  0.746812
>>> 2021-08-31  1.0  0.883054  0.739746
>>> 2021-09-01  1.0  0.879577  0.733322
>>> 2021-09-02  1.0  0.875766  0.725399
```

ewmLR

`pyg.timeseries._ewm.`**`ewmLR`** (a, b, n, time=None, min_sample=0.25, bias=True, axis=0, c=None, m=None, state=None) calculates pair-wise linear regression between a and b. We have a and b for which we want to fit:

```
>>> b_i = c + m a_i
>>> LSE(c,m) = \sum w_i (c + m a_i - b_i)^2
>>> dLSE/dc  = 0  <==> \sum w_i  (c + m a_i - b_i) = 0    [1]
>>> dLSE/dm  = 0 <==> \sum w_i  a_i (c + m a_i - b_i) = 0 [2]
```

```
>>> c     + mE(a)    = E(b)     [1]
>>> cE(a) + mE(a^2)  = E(ab)    [2]
```

```
>>> cE(a) + mE(a)^2  = E(a)E(n) [1] * E(a)
>>> m(E(a^2) - E(a)^2) = E(ab) - E(a)E(b)
>>> m = covar(a,b)/var(a)
>>> c = E(b) - mE(a)
```

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 in time unit.

- 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.

c,m : *place holder, ignore, optional*

ignore. The default is None.

state : *dict, optional*

Output from a previous run of ewmcor. The default is None.

Example: numpy arrays support

```
>>> assert eq(ewmLR(a.values, b.values, 10), ewmLR(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, 'tim
>>> assert abs(ts-df).max()<1e-10
```

Example: state management

```
>>> from pyg import *
>>> 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 = ewmLR_(old_a, old_b, 10)
>>> new_ts = ewmLR(new_a, new_b, 10, **old_ts) # instantiation with previous ewma
>>> ts = ewmLR(a,b,10)
>>> assert eq(new_ts.c, ts.c.iloc[5000:])
>>> assert eq(new_ts.m, ts.m.iloc[5000:])
```

Example:

```
>>> from pyg import *
>>> a0 = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> a1 = pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> b = (a0 - a1) + pd.Series(np.random.normal(0,1,10000), drange(-9999))
>>> a = pd.concat([a0,a1], axis=1)
>>> LR = ewmLR(a,b,50)
>>> assert abs(LR.m.mean()[0]-1)<0.5
>>> assert abs(LR.m.mean()[1]+1)<0.5
```

ewmGLM

`pyg.timeseries._ewm.`**`ewmGLM`** (a, b, n, time=None, min_sample=0.25, bias=True, data=None, state=None)

Calculates a General Linear Model fitting b to a.

Parameters:

a : a 2-d array/pd.DataFrame of values fitting b b : a 1-d array/pd.Series 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 in time unit.

- 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 the fitting. The default is 0.25. This ensures that we don’t get silly numbers due to small population.

data : *place holder, ignore, optional*

ignore. The default is None.

state : *dict, optional*

Output from a previous run of ewmGLM. The default is None.

Theory:

See https://en.wikipedia.org/wiki/Generalized_linear_model for full details. Briefly, we assume b is single column while a is multicolumn. We minimize least square error (LSE) fitting:

```
>>> b[i] =\sum_j m_j a_j[i]
>>> LSE(m) = \sum_i w_i (b[i] - \sum_j m_j * a_j[i])^2
```

```
>>> dLSE/dm_k = 0
>>> <==>  \sum_i w_i (b[i] - \sum_j m_j * a_j[i]) a_k[i] = 0
>>> <==>  E(b*a_k) = m_k E(a_k^2) + sum_{j<>k} m_k E(a_j a_k)
```

E is expectation under weights w. And we can rewrite it as:

```
>>> a2 x m = ab ## matrix multiplication
>>> a2[i,j] = E(a_i * a_j)
>>> ab[j] = E(a_j * b)
>>> m = a2.inverse x ab ## matrix multiplication
```

Example: simple fit

```
>>> from pyg import *
>>> a = pd.DataFrame(np.random.normal(0,1,(10000,10)), drange(-9999))
>>> true_m = np.random.normal(1,1,10)
>>> noise = np.random.normal(0,1,10000)
>>> b = (a * true_m).sum(axis = 1) + noise
```

```
>>> fitted_m = ewmGLM(a, b, 50)
```

ewmxo

`pyg.timeseries._ewmxo.`**`ewmxo`** (rtn, fast, slow, vol=None, instate=None)

This is the normalized crossover function

```
>>> res = (ewma(rtn, fast) - ewma(rtn, slow)) / (ewmstd(rtn, vol) * ou_factor(fast, slow))
```

The OU factor normalizes the result so that rms(res) is approximately 1

rtn: timeseries

The returns of a financial process

fast : *int/frac*

number of days. can also be 1/(1+days) if presented as a fraction

slow : *int/frc*

number of days. can also be 1/(1+days) if presented as a fraction

vol: int/frc

number of days. used for calculating the volatility horizon

Example:

```
>>> import numpy as np; import pandas as pd; from pyg import *
>>> rtn = pd.Series(np.random.normal(0,1,10000),drange(-9999,0))
>>> fast = 64; slow = 192; vol = 32; instate = None
```

functions exposing their state

simple functions

`pyg.timeseries._rolling.`**`diff_`** (a, n=1, axis=0, data=None, instate=None)

returns a forward filled array, up to n values forward. Equivalent to diff(a,n) but returns the full state. See diff for full details

`pyg.timeseries._rolling.`**`shift_`** (a, n=1, axis=0, instate=None)

Equivalent to shift(a,n) but returns the full state. See shift for full details

`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

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. 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. For full documentation, look at expanding_rms.__doc__

`pyg.timeseries._expanding.`**`expanding_std_`** (a, axis=0, data=None, instate=None)
Equivalent to expanding_mean(a) but returns also the state variables. 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. 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_mean(a) but returns also the state variables. 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. For full documentation, look at expanding_sum.__doc__

`pyg.timeseries._expanding.`**`cumprod_`** (a, axis=0, data=None, instate=None)
Equivalent to cumprod(a) but returns also the state variable. For full documentation, look at cumprod.__doc__

rolling window functions

`pyg.timeseries._rolling.`**`rolling_mean_`** (a, n, time=None, 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, time=None, 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, time=None, 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, time=None, 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, time=None, 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__

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.`**`ewmcorr_`** (a, n, min_sample=0.25, bias=False, instate=None)
This calculates a full correlation matrix as a timeseries. Also returns the recent state of the calculations. See ewmcorr for full details.

`pyg.timeseries._ewm.`**`ewmLR_`** (a, b, n, time=None, min_sample=0.25, bias=True, axis=0, c=None, m=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.`**`ewmGLM_`** (a, b, n, time=None, min_sample=0.25, bias=True, data=None, instate=None)
Equivalent to ewmGLM but returns a state parameter for instantiation of later calculations. See ewmGLM 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

Index handling

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,
>>> assert eq(df_fillna(df, 'ffill', limit = 2), np.array([np.nan,  1.,  1.,  1., np.nan,
```

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

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-
>>> 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
```

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])
```

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
```

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,
>>> assert eq(a(pd.Series([1,2,3,4], drange(-3)), np.array([1,2,3,4])),  pd.Series([2,4,6,
>>> assert eq(a(pd.Series([1,2,3,4], drange(-3)), np.array([[1,2,3,4],[5,6,7,8]]).T),  pd.
>>> 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]}, d
>>> 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)
```

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

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.

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.

initialization

```python
from pyg import *
Dict(a = 1, b = 2, c = 3)
```

```python
Dict(a = 1)(b = 2, c = 3)
```

```python
Dict(a = 1)(b = 2)(c = lambda a, b: a+b)
```

```python
Dict(a = 1) + dict(b = 2, c = 3)
```

members access

```python
d = Dict(a = 1, b = 2, c = 3)
```

```python
d.a
```

```python
d['a', 'b']
```

```python
d[['a', 'b']]
```

But the fun starts when Dict allows you to access **functions** of its keys:

```python
d[lambda a, b: a + b]
```

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.

```python
d = Dict({('a','b') : 1})
import pytest
with pytest.raises(KeyError): # Dict will be trying to grab 'a' and 'b' separately
    d[('a','b')]
```

adding

```python
Dict(a = 1, b = 2) + dict(b = 3, c = 4) # like .update() but not in-place
```

But addition is subtly different from update in the case of tree structure:

```python
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
```

Tree updating is actually important enough to have its own function that can operate on dict-trees

```python
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
```

subtracting

You can subtract keys or list of keys

```python
Dict(a = 'remove me', b = 2, c = 3) - 'a'  # subtracting a key
```

```python
Dict(a = 'I am gone', b = 'and so am I', c = 3) - ['a', 'b'] # subtracting a collection of k
```

```
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
```

modifying the keys: rename

```
Dict(a = 1, b = 2).rename('prefix_') # need to be done sufficient
```

```
Dict(a = 1, b = 2).rename('_suffix')
```

```
Dict(a = 1, b = 2).rename(upper)
```

```
Dict(a = 1, b = 2, c = 3).rename(a = 'Abraham', b = 'Barbara')
```

modifying the values: do

```
Dict(a = 1, b = 2, c = 3).do(lambda x: x**2) # modify all values using function
```

```
Dict(a = 1, b = 2, c = 3).do([lambda x: x**2, lambda x: x-1]) # modify all values using list
```

```
Dict(a = 1, b = 2, c = 3).do([lambda x: x**2, lambda x: x-1], 'a', 'b') # modify selected ke
```

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:

```
def func(a, b):
    c = a + b
    d = b + c
    e = a/b + d/c
    f = (d+e)/c
    return f
func(1,2)
```

How can we keep track of our calculations and debug it easily? Consider rewriting this:

```
x = Dict(a = 1, b = 2)
x = x(c = lambda a, b: a + b)
x = x(d = lambda b, c: b + c)
x = x(e = lambda a,b,c,d : a/b + d/c)
x = x(f = lambda c,d,e: (d+e)/c)
x
```

We have all the internals of the function exposed and we are able to separate calculation flow and data easily:

```
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)
```

```
initial_values(**calculation_pipeline)
```

You can see in pyg.base.dictable tutorial how this is extended

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.

Motivation: dictable as an organiser of research flow

We start with a simple motivating example. Here is a typical workflow:

```
from pyg import *; import pandas as pd; import numpy as np
import yfinance as yf
```

```
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]
```

```
zavgs
```

At this point we ask ourselves: Why do we have a **nan**? Which ticker was it, and when did it go wrong?

```
bad_symbols = [s for s, z in zip(symbols, zavgs) if np.isnan(z)]; bad_symbols
```

Great, how do we remove bad symbols from all our other variables?

```
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?

```
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.

Same code, in dictable

```
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())
```

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

```
s[['symbol', 'history', 'vol', 'zavg']]
```

```
s.zavg
```

Oh, no, we have a bad symbol, how do we remove it?

```
s = s.exc(zavg = np.nan); s.zavg
```

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:

```
sn = s * dict(n = [10,20,30]) ## each row is now unique per symbol and window n
```

```
sn = sn(ewma = lambda rtn, n, vol: rtn.ewm(n).mean()/vol)
```

And here is Citibank's three ewma...

```
sn.inc(symbol = 'C')[['n', 'ewma']]
```

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

```
sn.pivot('symbol', 'n', lambda ewma: ewma.mean())
```

dictable functionality

construction

dictable is quite flexible on constuctions.

```
d = dictable(a = [1,2,3,4], b = ['a', 'b', 'c', 'd']); d
```

```
d = dictable(dict(a = [1,2,3,4], b = ['a', 'b', 'c', 'd']), symbol = ['MSFT', 'AAPL', 'APA',
```

```
df = pd.DataFrame(d) # can instantiate a DataFrame from a dictable with no code and vice ver
```

```
d = dictable(df); d
```

```
d = dictable([(1,3), (2,4), (3,5)], ['a', 'b']); d # construction from records as tuples
```

```
d = dictable([dict(a = 1, b = 3), dict(a = 2, b = 4, d = 'new column'), dict(a = 3, b = 5, c
```

```
d = dictable(read_csv('d:/dropbox/yoav/python/pyg/docs/constituents_csv.csv')); d = d[:6]; d
```

row access

```
d[0] #returns a record
```

```
d[:2] ## subset rows using slice
```

```
for row in d: # iteration is by row
    print(row)
```

column access

```
d.Name
```

```
d['Name']
```

```
d['Name', 'Sector']
```

```
d[['Name', 'Sector']]
```

d is a dict so supports the usual keys(), values() and items():

```
for key, column in d.items():
    print(key, ':', column)
```

access via **function** of variables is also supported

```
d[lambda Symbol, Sector: '%s, %s'%(Symbol, Sector)]
```

column and row access are commutative

```
assert d[0].Name == d.Name[0] == '3M Company'
assert d[0][lambda Symbol, Sector: '%s, %s'%(Symbol, Sector)] == d[lambda Symbol, Sector: '%
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]
```

adding records

```
d = dictable(name = ['alan', 'barbara', 'chris'], surname = ['abramson', 'brown', 'cohen'],
```

```
d + dict(name = 'david', surname = 'donaldson', age = 4) ## adding a single record
```

```
d + [dict(name = 'david', surname = 'donaldson', age = 4), dict(name = 'evan', surname = 'em
```

```
d + dict(name = ['david', 'evan'], surname = ['donaldson', 'emmerson'], age = [4,5])
```

```
d + pd.DataFrame(dict(name = ['david', 'evan'], surname = ['donaldson', 'emmerson'], age = [
```

adding/modifying columns

You can add a column or a constant by simply calling the dictable with the values:

```
d(gender = ['m', 'f', 'm'])(school = 'St Paul')
```

More interestingly, it can be a callable function using the other variables…

```
d = d(initials = lambda name, surname: name[0] + surname[0]); d
```

Given d is a dict, a more traditional way of setting a new key is by simple assignment:

```
d['initials'] = d[lambda name, surname: name[0] + surname[0]]; d
```

Or you can use the dict.update method:

```
d.update(dict(gender = ['m', 'f', 'm'])); d
```

do

Sometime we want to apply the same function(s) to a collection of columns. For this, 'do' will do nicely:

```
d = d.do(upper, 'initials', 'gender').do(proper, 'name', 'surname'); d
```

removing columns

```
d = d - 'initials'; d
```

removing rows

```
d.exc(name = 'Alan')
```

```
d.inc(name = ['Alan', 'Chris'])
```

```
d.inc(lambda age: age>1)
```

```
d.exc(lambda gender: gender == 'M')
```

```
d.exc(lambda name, surname: len(name)>len(surname))
```

sort

```
d.sort('name', 'surname')
```

```
d.sort(lambda name: name[::-1]) # can sort on functions of variables too
```

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

```
grades = dictable(name = ['alan', 'barbara', 'chris'], grades = [30,90,80], subject = 'engli
       + dictable(name = ['alan', 'david', 'esther'], grades = [40,50,70], subject = 'math',
       + dictable(name = ['barbara', 'chris', 'esther'], grades = [90,60,80], subject = 'fre
```

```
grades.listby('teacher')
```

```
grades.listby('teacher')(avg_grade = lambda grades: np.mean(grades))
```

unlist

unlist undoes listby() assuming it is possible…

```
grades.listby('teacher').unlist()
```

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.

```
classes = grades.groupby(['teacher', 'subject'], grp = 'class')
```

```
classes[0]
```

```
classes.ungroup('class')
```

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

```
students = dictable(name = ['alan', 'barbara', 'chris', 'david', 'esthar', 'fabian'], surnam
```

```
print('shared keys:', grades.keys() & students.keys())
grades * students
```

Are there students with no surname? We can do a xor or use division which is overloaded for xor:

```
grades / students
```

Are there students with no grades?

```
students / grades
```

```
students = dictable(name = ['Alan', 'Barbara', 'Chris', 'David', 'Esther', 'Fabian'], surnam
```

We fixed Esther’s spelling but introduced capitalization, that is OK, we are allowed to inner join on functions of keys too.

```
grades.join(students, 'name', lambda name: name.lower())
```

```
students = dictable(first_name = ['alan', 'barbara', 'chris', 'david', 'esther', 'fabian'],
```

You can inner join on different column names and both columns will be populated:

```
grades.join(pd.DataFrame(students), 'name', 'first_name')
```

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

```
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
```

```
x.join(y, 'key', 'key', mode = 'left') ## grab left value
```

cross join

If no columns are shared, then a cross join is returned.

```
x = dictable(x = [1,2,3,4])
y = dict(y = [1,2,3])
x * y
```

```
x.join(y, [], []) ## you can force a full outer join
```

```
x / y == x
```

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

```
students = dictable(name = ['alan', 'barbara', 'chris'], surname = ['abramsom', 'brown', 'co
new_students = dictable(name = ['david', 'esther', 'fabian'], surname = ['drummond', 'eckles

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 availabl
left_join
```

```
# 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 stude
```

```
assert len(unmapped_grades / new_student_grades) == 0, 'students must exist either in the st
```

```
all_grades = student_grades + new_student_grades; all_grades
```

pivot

```
x = dictable(x = [1,2,3,4])
y = dictable(y = [1,2,3,4])
xy = (x * y)
xy
```

```
xy.pivot('x', 'y', lambda x, y: x*y)
```

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:

```
(xy + xy).exc(lambda x,y: x+y == 5).pivot('x', 'y', lambda x, y: x*y)
```

You can apply a sequence of aggregate functions:

```
(xy + xy).exc(lambda x,y: x+y == 5).pivot('x', 'y', lambda x, y: x*y, lambda v: len(v))
```

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

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:

```
{"$and": [{"name": {"$eq": "James"}}, {"surname": {"$eq": "Bond"}}]}
```

It's doable, but not much fun writing. Luckily… within the continuum you can write this instead:

```
from pyg import *; import re
q(name = 'James', surname = 'Bond')
```

```
(q.name == 'James') & (q.surname == 'Bond')
```

How do we create in MongoDB a query document to find all the James who are not Bond?

```
(q.surname!='Bond') & (q.name == 'James')
```

```
~(q.surname=='Bond') & (q.name == 'James')
```

What about records with no surname?

```
(q.name == 'James') - q.surname
```

```
q(q.surname.not_exists, name = 'James')
```

And what about records with james rather than James?

```
q(name = ['james', 'James'], surname = ['bond', 'Bond']) ## the result is long so it is repr
```

```
q(name = re.compile('^[J|j]ames'), surname = re.compile('^[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…

```
q['funny$text with # weird £ characters'].exists
```

If your document is nested and there are subkeys, that is ok, you can use either:

```
(q['key.subkey']>=100) | ((q.key.other.exists) & (q.some.other.stuff == [1,2]))
```

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…

```
bonds = dictable(name = ['Daniel', 'Sean', 'Roger', 'Timothy'], surname = ['Craig', 'Connery
bonds
```

```
q(list(bonds), q.release_date > dt(1980))
```

mongo_cursor

The mongo cursor:

- enables saving seemlessly objects and data in MongoDB

- simplifies filtering
- simplifies projecting onto certain keys in document

general objects insertion into documents

pymongo.Collection supports insertion of documents into it:

```
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
```

```
assert c.count_documents({}) == 1  # in order to count documents, must apply the empty query
```

We can do similar stuff with a mongo_cursor:

```
t = mongo_table(table = 'test', db = 'test')
t.drop()
t.insert_one(dict(a = 1, b = 2))
```

```
assert len(t) == 1 #no need to specify the filter, mongo_cursor keeps track of the current f
```

Annoyingly, raw pymongo.Collection cannot encode for lots of existing objects.

```
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

```
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:

```
t.drop()
t.insert_one(Dict(a = a, f = f, ts = ts, n = n))
assert len(t) == 1
t[0] ## reading it back
```

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:

```
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.

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
- 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
- Storing in files allows other non-python/non-MongoDB users easier access, allowing data to be detached from app. In particular, if you want to stream messages into the array/dataframe, doing it through Mongo is probably the wrong way about it. https://github.com/man-group/arctic attempts to do it but Mongo should probably just contain a reference to a file. You then have a listener such as 0MQ appending new messages into the file (perhaps via https://github.com/xor2k/npy-append-array/ or awswrangler). This will be (a) more performant, (b) require next to no code, and (c) new data will then magically show up in Mongo every time you read the document.

```
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)  ## c
t2.insert_one(doc)
encoded = c.find_one({})
print(tree_repr(encoded))
```

You can see that starting at the root location, the document's numpy arrays and pandas have been saved to .npy and .parquet files

```
print(tree_repr(decode(encoded)))
```

```
np.load('c:/temp/a/2.npy') ## can load data directly
```

```
pd_read_parquet('c:/temp/ts/one.parquet')
```

document access

We start by pushing a 10x10 times table into t

```
t.drop()
times_table = (dictable(a = range(10)) * dictable(b = range(10)))(c = lambda a, b: a*b)
t.insert_many(times_table)
```

filters

We now examine how we drill down to the document(s) we want:

```
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:

```
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]))
```

iteration

Just like a mongo.Cursor, c.find(spec), t is also iterable over the documents:

```
sum([doc for doc in t.find(a = 1).find(b = [1,2,3,4])], dictable())
```

```
dictable(t.find(a = 1).find(b = [1,2,3,4])) ## or just put a cursor straight into a table
```

```
t.find(a = 1).find(b = [1,2,3,4])[::] ## or simple slicing
```

sorting

```
t.sort('c', 'b')[::]
```

getitem of a specfic document

```
t[dict(a = 7, b = 8)]
```

column access

```
t.b
```

```
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:

```
t[['a', 'b']].find(c = 12)[::]
```

add/remove columns

```
del t['c']
t[::]
```

```
t = t.set(c = 'not very useful but...')
t[::]
```

```
t = t.set(c = lambda a, b: a * b) ### more useful
t[::]
```

add/remove records

```
t.inc(c = 12).drop()
t
```

```
t = t + dict(a = 2, b = 6, c = 12)
t
```

```
t = t.inc(c = 12).drop() + times_table.inc(c = 12) ## adding four records at once
t
```

```
t = t.inc(c = 12).drop().insert_many(times_table.inc(c = 12))
t[::]
```

```
t = t.raw ## remove the filter c = 12
t
```

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:

```
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', 'brow
pk.insert_many(d)
pk[::]
```

Now let us suppose a year has passed…

```
pk.set(age = lambda age: age + 1)
pk[::]
```

The pk-table actually maintains complete audit trail. Older records are not deleted, they just get '_deleted' parameter set for them.

```
print(dictable(c))
```

You can see pk only looks at records where _deleted does not exist and _pk are set.

```
pk
```

There are obvioursly 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

```
with pytest.raises(KeyError):
    pk.insert_one(dict(no_name_or_surname = 'James')) # cannot insert with no PK
```

```
pk.drop()
len(pk)
```

```
t[::] ## the data is there, it is just marked as _deleted
```

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:

```
pkr = mongo_table(table = 'test', db = 'test', pk = ['name', 'surname'], mode = 'r')
pkr
```

```
with pytest.raises(AttributeError):
    pkr.drop()
```

```
r  = mongo_table(table = 'test', db = 'test', mode = 'r')
with pytest.raises(AttributeError):
    r.drop()
```

```
r[::]
```

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

Cell 101

```
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
```

```
b.keys() ## b is a dict
```

```
b._args ## inputs
```

```
b._output ## where the output will go once we calculate it
```

```
assert b.run() ## b has not calculated yet... please run it
```

```
b() # calculated object note b().data
```

```
assert not b().run() ## b has calculated now... no need to run it
```

```
cell(lambda x, y: x ** y)(x = a, y = 2) # you can define the cell and then call it with the
```

Workflow without saving to the database

```
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')).re
constituents
```

```
stocks = constituents.inc(sector = 'Energy')
stocks
```

```
stocks = stocks(history = lambda symbol, sector, name: yf.download(tickers = symbol))
```

```
stocks = stocks.inc(lambda history: len(history)>0)
```

```
stocks = stocks(adj = lambda history: getitem(value = history, key = 'Adj Close'))
```

```
stocks = stocks(rtn = lambda adj: diff(a = adj))
stocks = stocks(vol = lambda rtn: ewmstd(a = rtn, n = 30))
```

```
_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_.s

crossover_.output = ['data', 'state']

def crossover(a, fast, slow, vol, state = None):
    return crossover_(a, fast, slow, vol, instate = state)
```

some more functions to calculate the profits & loss as well as the signal/noise ratio

```
def signal_pnl(signal, rtn, vol):
    return shift(signal) * (rtn/vol)

def information_ratio(pnl):
    return 16 * ts_mean(pnl) / ts_std(pnl)
```

```
forecasts = stocks * dictable(fast = [2,4,8], slow = [6,12,24], forecast = ['fast', 'medium'
```

```
forecasts = forecasts(signal = lambda rtn, fast, slow: crossover_(rtn, fast = fast, slow = s
```

```
forecasts = forecasts(pnl = lambda signal, rtn, vol: signal_pnl(signal = signal, rtn = rtn,
```

```
forecasts = forecasts(ir = lambda pnl: information_ratio(pnl = pnl))
```

```
print(forecasts.pivot('symbol', 'forecast', 'ir', [last, f12]))
```

Workflow while saving to MongoDB

Table creation

We create three tables dependending on the primary keys we will be using.

```
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', 'forecas
```

```
idb().insert_one(Dict(item = 'constituents', data = constituents))
```

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.

```
stocks = stocks(history = lambda symbol, sector, name: periodic_cell(yf.download, tickers =
                                              db = sdb, item = 'history', symbol = symbol)())
```

Accessing the data in MongoDB

The data is now in the database and can be accessed:

```
get_data('stock', 'demo', symbol = 'MMM')
```

```
stocks = stocks.inc(lambda history: len(history.data)>0)
```

```
stocks = stocks(adj = lambda history, symbol: periodic_cell(getitem, value = history, key =
                                                              db = sdb, symbol = symbol, item
```

```
stocks = stocks(rtn = lambda adj, symbol: periodic_cell(diff, a = adj,
                                                          db = sdb, symbol = symbol, item = 'r
```

```
stocks = stocks(vol = lambda rtn, symbol: periodic_cell(ewmstd, a = rtn, n =  30,
                                                          db = sdb, symbol = symbol, item = 'v
```

Calculating the forecasts & saving them

```
forecasts = stocks * dictable(fast = [2,4,8], slow = [6,12,24], forecast = ['fast', 'medium'
```

```
forecasts = forecasts(signal = lambda rtn, fast, slow, symbol, forecast: periodic_cell(cross
                                                  db = fdb, symbol = symbol, forecast = forecast,
```

```
forecasts = forecasts(pnl = lambda signal, rtn, vol, symbol, forecast: periodic_cell(signal_
                                                          db = fdb, symbol = symbol, forecast
```

```
forecasts = forecasts(ir = lambda pnl: information_ratio(pnl = pnl.data))
```

```
print(forecasts.pivot('symbol', 'forecast', 'ir', [last, f12]))
```

Accessing & running the graph once the graph has been created

We can access the data or the cell:

```
get_cell('forecast', 'demo', symbol = 'APA', forecast = 'fast', item = 'signal')
```

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:

```
c = get_cell('forecast', 'demo', symbol = 'APA', forecast = 'fast', item = 'signal')
c = c.go()
print(c.data)
```

Point-in-time, cache and persistency

The pk-tables save a full history of all your data. To avoid hitting the database all the time, we also have a local GRAPH singleton that caches all the cells by their address. The cell has few basic operations we need to understand:

- **cell.run()**: Returns True/False if the cell needs to be calculated. db_cell() just check for values in its output, periodic_cell will also check if it is a new business day.
- **cell.go()**: This calculates the cell and saves the result to the database (and to GRAPH). The cell itself is not loaded but all its inputs are loaded
    - cell.go(0) : calculate only if there is a need

    - cell.go(1) : calculate me but my parents only if there is a need
    - cell.go(2) : calculate me & my parents but my grandparents only if there is a need
    - cell.go(-1) : calculate everything

- **cell.load()**: This loads the data from GRAPH, if not in GRAPH, loads it from MongoDB (and updates also the GRAPH)

    - cell.load(-1) : Clear the data from the GRAPH
    - cell.load(0) : Load & update me from GRAPH, if not, from MongoDB, if not, just return good old me
    - cell.load(1) : If you cannot find me, throw an Exception
    - cell.load(date) : Load my version as valid on date. If none exists, throw.
    - cell.load([value]): Force GRAPH to clear, only load from DB. Same as cell.load(-1).load(value)

- **cell()**: This loads & then go

```
from pyg import *; from functools import partial
db = partial(mongo_table, db = 'demo', table = 'persistency', pk = 'key')
db().raw.drop()

def f(a, b):
    return a+b
```

```
## now we set up a fake calculation tree:
x = db_cell(f, a = 1, b = 2, db = db, key = 'x')
y = db_cell(f, a = x, b = 2, db = db, key = 'y')
z = db_cell(f, a = x, b = y, db = db, key = 'z')

## and run it by running the final value we want
z = z()
```

```
## we can access the data:
get_data('persistency', 'demo', key = 'x')
```

```
t0 = dt()  ## first breakpoint
```

```
x = db_cell(f, a = 10, b = 20, db = db, key = 'x').go()
y = db_cell(f, a = x,  b = 20, db = db, key = 'y').go()
z = db_cell(f, a = x,  b = y, db = db, key = 'z').go()
```

```
## and here is the new data
get_data('persistency', 'demo', key = 'x')
```

```
## and here is the data valid at our first breakpoint. get_data/get_cell always go to the da
get_data('persistency', 'demo', key = 'x', _deleted = t0)
```

We can ask a cell to load itself, but remember: it will go to GRAPH first by default. The GRAPH has only one copy of the cell, while in MongoDB, every time we recalculate/save a new version of the cell, we mark the old version in the database as “deleted” but otherwise keep it. To force a cell to load itself from the database, encase the breakpoint in a list…

```
db_cell(db = db, key = 'x').load([t0])
```

We can force a full recalculation of the tree in a single line of code:

```
db_cell(db = db, key = 'z')(go = -1, mode = [t0]).data ## Should be 8, same as the old value
```

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 (and indeed, rerun) 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)

To save or not to save?

Luckily we don't really need to decide on one workflow or the other as both can happily coexist. We can build a calculation graph and decide that some key points in the calculation we want to save while intermediate calculations we can calculate on the fly and not save at all. We have met the crossover function. Here we implement it 'on the fly' while saving just final value to db

```
from pyg import *; import pandas as pd; import numpy as np; from functools import partial
```

```
def fake_ts(ticker):
    return pd.Series(np.random.normal(0,1,1000), drange(-999))
db = partial(mongo_table, db = 'test', table = 'test', pk = ['key'])
db().raw.drop()
```

```
appl = db_cell(fake_ts, ticker = 'appl', key = 'appl_rtn', db = db)()

#I am never saving these, In fact, I don't want to see these in calculation log.
a = cell(ewma, a = appl, n = 30)
b = cell(ewma, a = appl, n = 50)

# I may want to save these nodes but haven't made up my mind
# I do want to see the calculations in the log though...
# I replace db by the primary keys of table (here 'key').
# This allows as to see the calculation log as it happens.
# data is not saved to db though until I switch to db = db as opposed to db = 'key'

c = db_cell(sub_, a = a, b = b, key = 'calculate difference of ewma', db = 'key')
d = db_cell(ewmrms, a = c, n = 100, key = 'root mean square of difference', db = 'key')

# The final crossover I definitely want to save to db:
final_value = db_cell(div_, a = c, b = d, key = 'appl_crossover', db = db)()
```

```
db().key
```

So although we had several intermediate steps, we decided to save just the final crossover in the database, If we look at the inputs for the function, you can see that the values are not saved in the database, though the full calculation tree *is*. Therefore, one can reload the node and then recalculate all the intermediate values on the fly

```
loaded_and_recalculated = (db()[dict(key = 'appl_crossover')] - 'data').go(1) ## but once re
assert eq(loaded_and_recalculated.data, final_value.data)
```

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…

```
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.

```
a = cell(data = 1)
cell(twox, x = a)()
```

‘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:

```
cell_output(a)
```

```
a = cell(data = 1, myoutput = 3, output = 'myoutput') ## you can decide your output is diffe
cell_output(a), cell_item(a)
```

```
cell(twox, x = a)()
```

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?

```
a = cell(a = 1) ## this has failed...
cell(cell_func(twox, x = 'a'), x = a)() ## when you grab x, use 'a' as key
```

What if you need the cell itself rather than the items in it?

```
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)()
```

We can see that the cell x itself is presented to the function and x.a + x.b is calculated and data == 3

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.

Join

Example: Using join function to transfer money to a bank

We begin by setting up a mini database:

```
from pyg import *
customers = dictable(customer = ['alan', 'barbara', 'charles'], address = ['1 Abba Avenue',
products = dictable(product = ['apple', 'banana', 'cherry'], price = [1,2,3], supplier  = ['
customer_products = dictable(customer = ['alan', 'alan', 'charles', 'charles'], product = ['
banks = dictable(bank = ['allied', 'barclays'], account = [5556, 2461])

print('Customers\n', customers, '\n\nProducts\n', products, '\n\nCustomer_products\n', custo
```

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:

```
join(dict(bank = customers, amount = customer_products, price = products), on = ['customer',
```

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.

```
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:

```
join(dict(bank = customers, amount = customer_products, price = products, account = banks),
```

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….

```
join(dict(bank = customers, amount = customer_products, price = products, account = banks),
     on = ['customer',  'product', 'bank'],
     defaults = dict(account = 'default'))
```

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

```
join(dict(bank = customers, amount = customer_products, price = products, account = banks, e
     on = ['customer',  'product', 'bank'],
     renames = dict(expiry = lambda purchase_date: dt(purchase_date, '1b')),  ## it takes 1
     defaults = dict(account = 'default'))
```

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.

```
from pyg import *
oil = dictable(y = dt().year-1, m = range(3, 13, 3)) + dictable(y = dt().year, m = range(3,
oil = oil(ticker = lambda y, m: 'OIL_%i_%s'%(y, m if m>9 else '0%i'%m))
oil
```

y,m and ticker will form our primary keys

```
pk = ['y', 'm', 'ticker']
expiry = perdictable(lambda y, m: dt(y,m+1,1), on = pk)(y = oil, m = oil)
expiry
```

```
def fake_ts(ticker, expiry):
    return 500 + pd.Series(np.random.normal(0,1,100), drange(dt_bump(expiry,-99), expiry)).c
```

To add a price for each of the futures, we first wrap fake_ts and then run it:

```
price = perdictable(fake_ts, on = pk)(ticker = oil, expiry = expiry)
price
```

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

```
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
```

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.

```
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')
```

perdictable with the cell framework

We can run the function and use a cell to store the output…

```
c = cell(perdictable(fake_ts, on = pk), ticker = oil, expiry = expiry)()
c.data
```

```
recalculated_cell = c.go(1) ## force a recalculation
recalculated_cell.data
```

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.

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:

```
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 a
```

If you want to see the full calculations and inputs to the function set **include_inputs**=True:

```
perdictable(function, on = 'product', include_inputs = True)(price = price, quantity = quant
```

If you want output column to be not data, use **col**:

```
perdictable(function, on = 'product', include_inputs = True, col = 'cost')(price = price, qu
```

The **if_none** parameter determines how data is calculated for rows that have expired but their data is None:

```
expiry = dictable(product = ['apple', 'banana', 'cherry'], expiry = [dt(-2), dt(-1), dt(1)])
previous_data = dictable(product = ['apple', 'banana', 'cherry'], data = [None, 'some value
perdictable(function, on = 'product', include_inputs = True, if_none = False)(price = price,
```

```
perdictable(function, on = 'product', include_inputs = True, if_none = True)(price = price,
```

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.

```
def running_total_costs(price, quantity=1, data=0):
    return data + price * quantity
```

```
previous_data = dictable(product = ['apple', 'banana', 'cherry', 'damson'], data = [10, 20,
perdictable(running_total_costs, on = 'product', include_inputs = True)(price = price, quant
```

```
## 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 = Fa
```

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.

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.

Agreement between pyg.timeseries and pandas

```
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))
```

```
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
```

```
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
```

```
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
```

```
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 ##
```

Quick performance comparison

pyg, when run on pandas dataframes rather than arrays, is of comparable speed to pandas

```
compare = dictable(op =  ['rolling_sum', 'rolling_mean', 'rolling_std', 'rolling_min', 'roll
             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(1

compare += dictable(op =  ['expanding_sum', 'expanding_mean', 'expanding_std', 'expanding_mi
             pyg = [expanding_sum, expanding_mean, expanding_std, expanding_min, expanding_m
             pandas = [s.expanding().sum, s.expanding().mean, s.expanding().std, s.expanding

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,

print(compare(winner = lambda pyg, pandas: 'pyg' if pyg<pandas * 0.8 else 'pandas' if pyg >
```

pyg and numpy arrays

pyg supports numpy arrays natively. Indeed, pyg is 3-5 times faster on numpy arrays.

```
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
```

```
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
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
```

```
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
```

pandas treatment of nan

Suppose we have weekly data that at some point we resample to daily… The two look the same…

```
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)
```

… but any calculation using the daily will yield a different result from a calculation on the weekly which is then resampled to daily:

```
pd.concat([weekly.ewm(4).mean().reindex(days), daily.ewm(4).mean()], axis = 1) ## The result
```

```
pd.concat([weekly.diff().reindex(days), daily.diff()], axis = 1) ## The result depends on wh
```

Indeed, for diff, daily.diff() is all nan!

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.

```
nona(pd.concat([ewma(weekly, 4).reindex(days), ewma(daily,4)], axis = 1)) ## The two match e
```

```
nona(pd.concat([diff(weekly).reindex(days), diff(daily)], axis = 1)) ## The result depends o
```

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.

```
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
```

```
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
```

```
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
joint_signal[dt(0):]
```

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.

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:

```
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

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
```

We can quickly rewrite it using pyg:

```
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
```

And with very little additional effort, we can write a new function that also exposes the internal state:

```
_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_.s

crossover_.output = ['data', 'state'] # output declares the function to have a dict output a

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
```

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:

```
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?

```
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
```

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.

```
scenarios = np.random.normal(0,1,(100,10000))
x_scenarios = timer(crossover)(scenarios , 10, 30, 50, state = x_history.state)
```

Using cells, our code looks like this, with live and historical codebase looking pretty similar

```
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
```

```
pd.concat([pd.Series(x_live.data, pandas_x.index[-100:]), pandas_x.iloc[-100:]], axis = 1)
```

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.

```
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.

```
a = np.arange(5)
sumsq(a)
```

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.

```
s = pd.Series(a, drange(-4))
sumsq(s)
```

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:

```
@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
```

```
sumsq(df)
```

Indeed, since we asked it to loop over dict, list and numpy array (2d)

```
sumsq(dict(a = a, b = a+1))
```

```
sumsq(df.values)
```

presync: manage indexing and date stamps

Suppose the function takes two (or more) timeseries.

```
@presync(index = 'inner')
@loop(pd.DataFrame, np.ndarray)
@pd2np
```

```
def product(a, b):
    return a * b
```

```
a = np.arange(5); b = np.arange(5)
product(a,b)
```

What happens when the weights and the timeseries are unsynchronized?

```
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)
```

```
product(a_, b_) ## just the inner values
```

```
product.oj(a_, b_) ## outer join
```

```
product.oj.ffill(a_, b_) ## outer join and forward-fill
```

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.

```
a = np.arange(5); b = np.arange(1,5)
a, b
```

```
product(a, b)
```

```
us = calendar('US')
dates = pd.Index(us.drange('-40y', 0 ,'1b'))
```

```
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,
universe
```

```
universe = universe(rtn = lambda ts: ts.values)
universe = universe(price = lambda rtn : cumsum(rtn))
universe = universe(vol = lambda rtn: ewmstd(rtn, 30))
universe
```

```
presync(lambda tss: np.array(tss).T)(universe.vol)
```

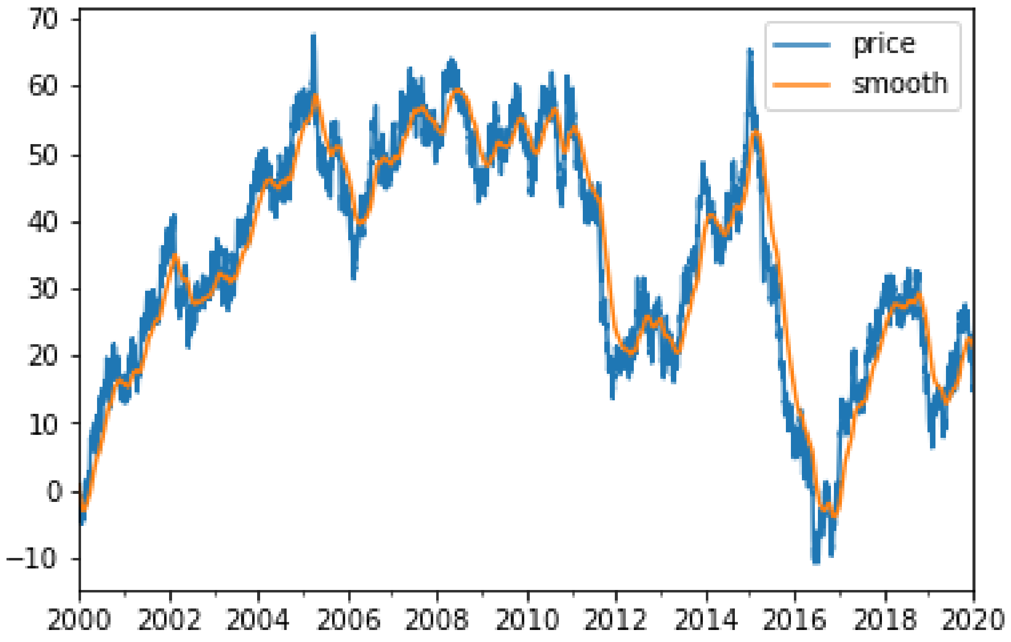
```
universe = universe.do(lambda value: np_reindex(value, dates), 'rtn', 'price', 'vol')
universe
```

```
vol = pd.concat(universe.vol, axis = 1); vol.columns = universe.stock
vol
```

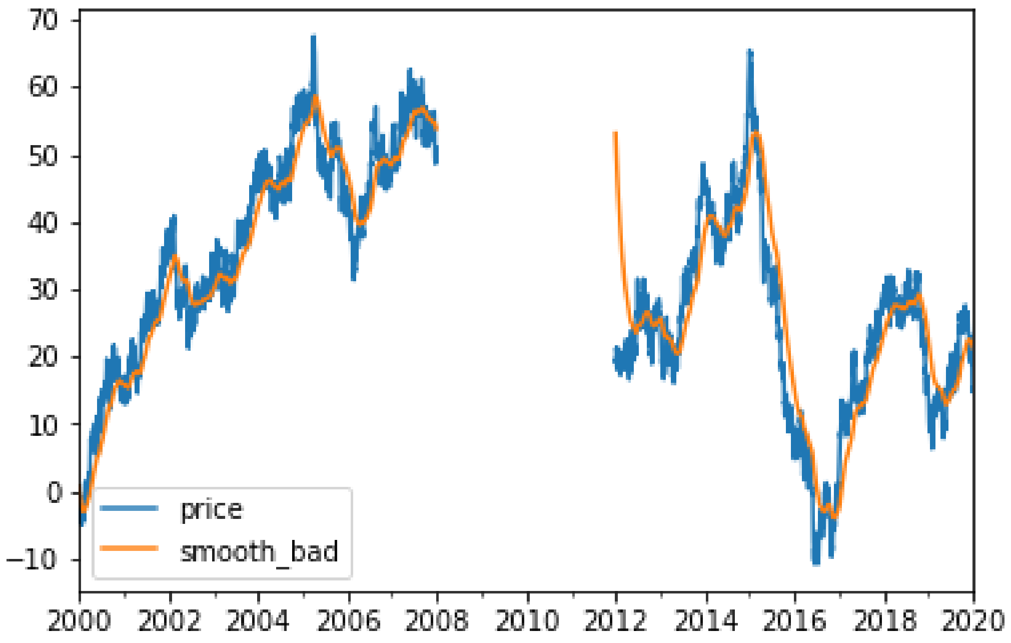
pyg.timeseries.ewma

The ewm functions implement the concept of time which we think is worthwhile explaining. We start with an example

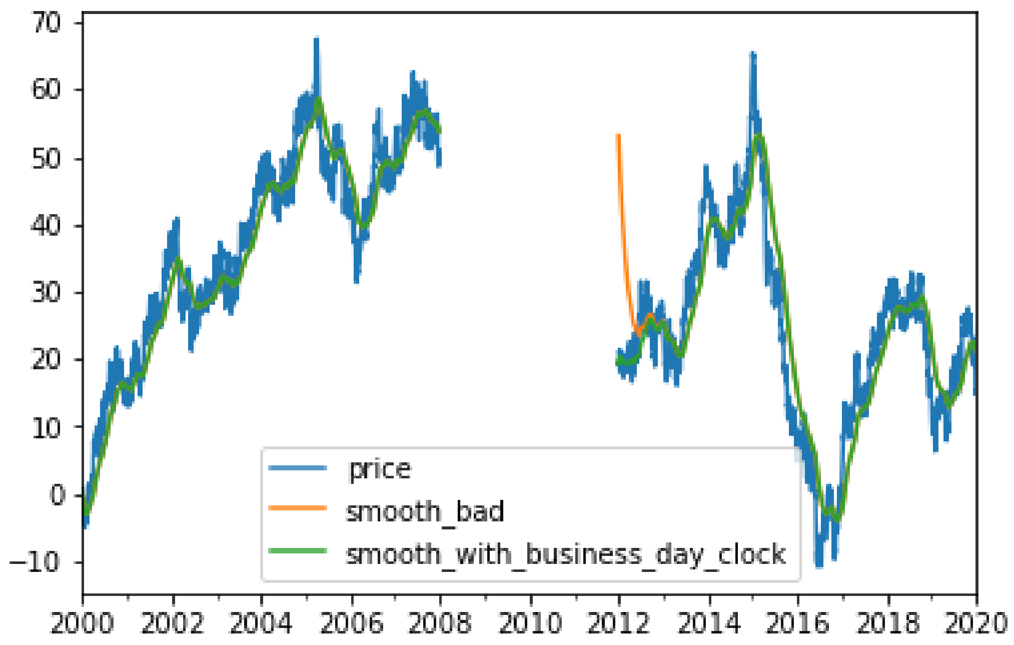
```
from pyg import *; import numpy as np; import pandas as pd
rtn = pd.Series(np.random.normal(0.01,1,5218), drange(2000,2020, '1b'))
price = cumsum(rtn); price.name = 'price'
smooth = ewma(price, 50); smooth.name = 'smooth'
pd.concat([price, smooth], axis = 1).plot()
```



```
## 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()
```



```
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()
```

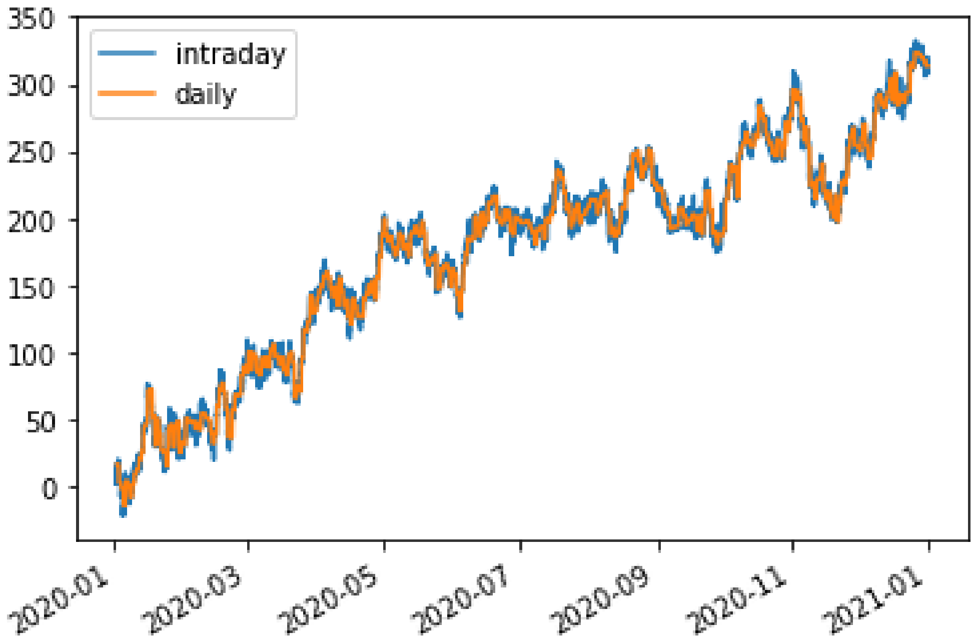


What happened here? How can smooth with clock track better? The answer is that if you provide a clock, ewma can recognise that 4 years have passed. The old data is irrelevant, it forgets the old position and start with most of the weight on the more recent observations

What happens if the clock does not move at all?

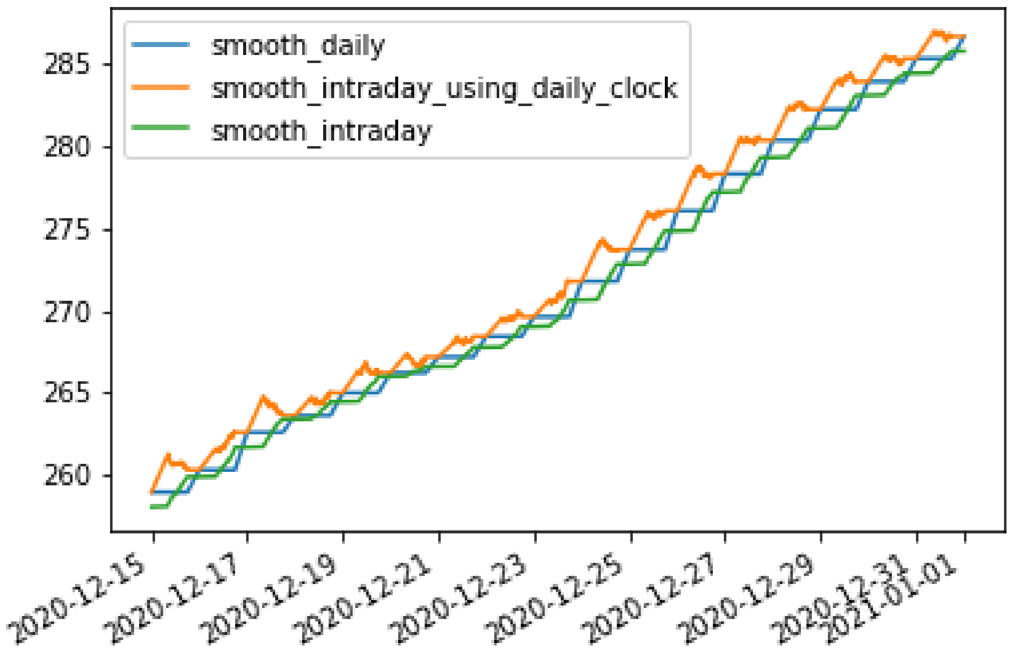
Suppose we now choose to calculate daily ewma but we are doing this with intraday data. If the number of data points per day is constant and known, then this can be done with ease. Using time parameter, we can do this even for an irregularly spaced timeseries. We first just create some fake data:

```
import datetime
bar = datetime.timedelta(minutes = 5)
all_bars = [t for t in drange(2020, 2021, bar)]
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
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()
```



- 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 througut the day

```
smooth_intra = ewma(price, 20, 'f'); smooth_intra.name = 'smooth_intraday' ## roughly matchi
smooth_intra_using_d = ewma(price, 20, time = 'd'); smooth_intra_using_d.name = 'smooth_intr
smooth_daily = ewma(daily, 20); smooth_daily.name = 'smooth_daily'
pd.concat([smooth_daily, smooth_intra_using_d, smooth_intra], axis = 1).ffill()[dt(2020,12,1
```



- 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

```
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
```

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

Indices and tables

- **`genindex`**
- **`modindex`**
- **`search`**

Python Module Index