VectorDict Documentation

Release 1.0.0

Julien Tayon

May 24, 2012

CONTENTS

- source : https://github.com/jul/ADictAdd_iction/
- online documentation : http://readthedocs.org/docs/vectordict/en/latest/
- ticketing : https://github.com/jul/ADictAdd_iction/issues

CHAPTER

ONE

MANIPULATING DICT OF DICT AS TREES PRETTY EASILY

Contents:

1.1 What's new in 1.0.0

1.1.1 API Change (backward compatible with 0.6.0)

converter() now takes a second argument which defaults to int that is used as a default factory for the underlying defaultdict

path_to_tree() has the same change.

1.1.2 Boolean operations

These are kept for backward compatibility but not documented anymore. Commutation don't work. Refactoring is needed.

1.2 What's new in 0.6.0

1.2.1 API Change

formating

- for the sake of consistency added a pformat for the pprint
- change keys value separator in tformat to **:**.

1.2.2 not is a problem

Can't find how to overload *! atree*. *__not__* is **not** behaving the way I want. I cannot write *a ^ b = (a & (!b)) | ((!a) & b)*

Their will be an inconsistency in the logical operation **notation** because of this pecular side of OO programming as considered right by python specifications.

Planning on using **rand / ror / rnot / rxor** in the meaning of recursive ... operations.

I should change all methods naming according to this. Least surprise principle they said :(it is pretty surprising to me that operator overloading cannot be done when all is object... and that it work 99% of the time.

Too much of a headache write now I postpone thinking

1.2.3 future API change for logical operations

or behaviour mixes its behaviour with union and or, and **and** with intersection and logical and. As a dramatic result : **xor** does not give the same results as **not xand**. So I will decide either to standardize a three value boolean algebrae where

- value & undefined = False
- value | undefined = value

which would be equivalent to state **None = False** but python says :

```
>>> None == False
False
>>> None is False
False
```

or

- if not defined value, dont guess and I'll prune all paths that are not define

or

- raise **TypeError Exception** for unsupported type (empty path vs known path)

1.2.4 Almost complete support for set operation

We have a problem inherent with python not being able to tell if two functions shares same code and context. func1 == func2 is synonym they are in t

So set operations will fail in case you use functions.

1.3 Vector Dict

Manipulating dict of dict as trees with :

- set operations
- logical operations
- tree manipulations
- search operation
- metrics of similarities
- vector algebrae

1.3.1 Helpers

class vector_dict.VectorDict.**tree_from_path**
creating a dict from a path

```
>>> tree_from_path( 'a', 'b', 'c', 1, defaultdict_factory = int  ).tprint()
{
    a : {
        b : {
            c : 1,
        },
    },
}
```

class vector_dict.VectorDict.**convert_tree**

convert from any other nested object to a VectorDict espaecially usefull for constructing a vector dict from intricated dict of dicts. *Dont work as a class method (why ?)*

```
>>> convert_tree({ 'a' : { 'a' : { 'r' : "yop", 'b' : { 'c' :  1 }}}}).tprint()
{
    a : {
        a : {
            r : 'yop',
            b : {
                c : 1,
            },
        },
    },
}
```

** BUG : if empty dict is a leaf, default_factory is not applied** workaround :you can specify Krut as leaves explicilty let's define 3 domain :

- •root is defaulting to str
- •a is defaulting to list
- •b defaulting to int

```
>>> from vector_dict.VectorDict import VectorDict as krut
>>> from vector_dict.VectorDict import converter as kruter
>>> a = kruter( { 'a' : Krut( list, {} ),'b' : Krut(int, {}) }, str )
>>> a['b']['e'] += 1
>>> a['a']['c'] += [ 3 ]
>>> a['d'] += "toto"
>>> a.tprint()
{
    'a' : {
        'c' : [3],
    },
    'b' : {
        'e' : 1,
    },
    'd' : 'toto',
}
```

class vector_dict.VectorDict.**VectorDict**(**a*, ***a_dict*)
slightly enhanced Dict

tprint(*indent_level=0*, *base_indent=4*)
pretty printing with indentation in tradiotionnal fashion

pprint()
pretty printing the VectorDict in flattened vectorish representation

tformat(*indent_level=0*, *base_indent=4*)
pretty printing in a tree like form a la Perl

1.3.2 Set Operations

class vector_dict.VectorDict.**VectorDict**(**a*, ***a_dict*)
slightly enhanced Dict

intersection(*other*)
Return all elements common in two different trees raise an exception if both leaves are different

#TOFIX : dont make a newdict if doing in place operations

```
>>> from vector_dict.VectorDict import convert_tree, VectorDict,Element,Path
>>> a = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, c = 2  ) ) } )
>>> b = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, d = 1  ) ) } )
>>> a.intersection(b).tprint()
{
    a : {
        b : 1,
    },
}
>>> b = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, c = 1  ) ) } )
>>> a.intersection(b).tprint()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "vector_dict/VectorDict.py", line 634, in intersection
    new_dict[k] = (self[k]).intersection( other[k] )
  File "vector_dict/VectorDict.py", line 639, in intersection
    other[k]
Exception: ('CollisionError', '2 != 1')
```

symmetric_difference(*other*)
return elements present only elements in one of the dict Throw a Collision Error if two leaves in each tree are different

```
>>> from vector_dict.VectorDict import convert_tree, VectorDict,Element,Path
>>> a = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, d = 2, c=1  ) ), 'e' :  1 } )
>>> b = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, c = 1   ) ) } )
>>> a.symmetric_difference(b)
defaultdict(<type 'int'>, {'a': defaultdict(<type 'int'>, {'d': 2}), 'e': 1})
>>> b = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, c = 2   ) ) } )
>>> a.symmetric_difference(b)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "vector_dict/VectorDict.py", line 694, in symmetric_difference
    for path,v in self.intersection(other).as_vector_iter():
  File "vector_dict/VectorDict.py", line 658, in intersection
    new_dict[k] = (self[k]).intersection( other[k] )
  File "vector_dict/VectorDict.py", line 663, in intersection
    other[k]
Exception: ('CollisionError', '1 != 2')
```

union(*other*, *intersection=None*)
return the union of two dicts

```
>>> from vector_dict.VectorDict import convert_tree, VectorDict,Element,Path
>>> b = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, c = 1  ) ) } )
>>> a = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, d = 2, c=1  ) ), 'e' :  1 } )
>>> b.union(a)
defaultdict(<type 'int'>, {'a': defaultdict(<type 'int'>, {'c': 1, 'b': 1, 'd': 2}), 'e': 1}
>>> a = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, d = 2, c=3  ) ) } )
>>> b.union(a)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "vector_dict/VectorDict.py", line 669, in union
    return self + other - self.intersection(other)
  File "vector_dict/VectorDict.py", line 658, in intersection
    new_dict[k] = (self[k]).intersection( other[k] )
  File "vector_dict/VectorDict.py", line 663, in intersection
    other[k]
Exception: ('CollisionError', '1 != 3')
```

issubset(*other*)
tells if all element of self are included in other

```
>>> from vector_dict.VectorDict import convert_tree, VectorDict,Element,Path
>>> a = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, d = 2, c=1  ) ), 'e' :  1 } )
>>> b = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, c = 1  ) ) } )
>>> b.issubset(a)
True
>>> a.issubset(b)
False
```

issuperset(*other*)
tells if all element of other is included in self throws an exception if two leaves in the two trees have different values

```
>>> from vector_dict.VectorDict import convert_tree, VectorDict,Element,Path
>>> a = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, d = 2, c=1  ) ), 'e' :  1 } )
>>> b = VectorDict( int, { 'a' : VectorDict( int, dict( b = 1, c = 1  ) ) } )
>>> a.issuperset(b)
True
>>> b.issuperset(a)
False
```

1.3.3 Iterators

class vector_dict.VectorDict.**VectorDict**(**a*, ***a_dict*)
slightly enhanced Dict

as_vector_iter(*path=()*)

iterator on key value pair of nested dict in the form of set(key0, key1, key2), child for a dict, therefore making a n-depth dict being homomorph to a single dimension vector in the form of k , v where k is the path, v is the leaf value source: http://tech.blog.aknin.name/2011/12/11/walking-python-objects-recursively/

```
>>> a = convert_tree({ 'a' : { 'a' : { 'r' : "yop" , 'b' : { 'c' :  1 }}}})
>>> a.tprint()
{
```

```
        a = {
            a = {
                r = 'yop',
                b = {
                    c = 1,
                },
            },
        },
    }
    >>> [ e for e in a.as_vector_iter() ]
    [(('a', 'a', 'r'), 'yop'), (('a', 'a', 'b', 'c'), 1)]
```

as_row_iter(*path=()*, ***arg*)

iterator on key value pair of nested dict yielding items in the form set(key0, key1, key2 , child) very useful for turning a dict in a row for a csv output all keys and values are flattened

```
>>> a = convert_tree({ 'a' : { 'a' : { 'r' : "yop" , 'b' : { 'c' :  1 }}}})
>>> a.tprint()
{
    a : {
        a : {
            r : 'yop',
            b : {
                c : 1,
            },
        },
    },
}
>>> [ e for e in a.as_row_iter() ]
[['a', 'a', 'r', 'yop'], ['a', 'a', 'b', 'c', 1]]
```

1.3.4 Accessing and modifying

class vector_dict.VectorDict.**VectorDict**(**a*, ***a_dict*)

slightly enhanced Dict

at(*path*, *apply_here=None*, *copy=False*)

gets to the mentioned path eventually apply a lambda on the value and return the node, and copy it if mentioned.

```
>>> intricated = convert_tree( { 'a' : { 'a' : { 'b' : { 'c' :  1 } } } } )
>>> pt = intricated.at( ( 'a', 'a', 'b' ) )
>>> pt
defaultdict(<class 'vector_dict.VectorDict.VectorDict'>, {'c': 1})
>>> pt['c'] = 2
>>> intricated.tprint()
{
   a : {
       a : {
           b : {
               c : 2,
           },
       },
   },
}
>>> intricated.at( ( 'a', 'a', 'b' ), lambda x : x * -2  )
```

```
defaultdict(<class 'vector_dict.VectorDict.VectorDict'>, {'c': -4})
>>> intricated.pprint()
a->a->b->c = -4
```

get_at(**path*)
Get a copy of an element at the coordinates given by the path Throw a KeyError excpetion if the path does not led to an element

```
>>> from vector_dict.VectorDict import convert_tree, VectorDict
>>> intricated = convert_tree( { 'a' : { 'a' : { 'b' : { 'c' :  1 } } } } )
>>> intricated.get_at( 'a', 'a', 'b' )
defaultdict(<class 'vector_dict.VectorDict.VectorDict'>, {'c': 1})
>>> intricated.get_at( 'a', 'a', 'b', 'c' )
1
>>> intricated.get_at( 'oops' )
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "vector_dict/VectorDict.py", line 304, in get_at
    return self.at( path, None , True)
  File "vector_dict/VectorDict.py", line 330, in at
    value = here[path[ -1 ] ]
KeyError: 'oops'
```

prune(**path*)
delete all items at path

```
>>> a  = VectorDict(int, {})
>>> a.build_path( 'g', 'todel' , True )
>>> a.build_path( 'g', 'tokeep' , True )
>>> a.tprint()
{
    g : {
        tokeep : True,
        todel : True,
    },
}
>>> a.prune( 'g', 'todel' )
>>> a.tprint()
{
    g : {
        tokeep : True,
    },
}
```

find(**a*, ***kw*)
apply a fonction on value if predicate on key is found

build_path(**path*)
implementation of constructing a path in a tree, argument is a serie of key

```
>>> a = VectorDict(int, {})
>>> a.build_path( 'k', 1 )
>>> a.tprint()
{
    k = 1,
}
>>> a.build_path( 'b', 'n', [ 1 ] )
>>> a.build_path( 'd', 'e',  2  )
>>> a.build_path( 'd', 'f',  4  )
```

```
>>> a.tprint()
{
    k : 1,
    b : {
        n : [1],
    },
    d : {
        e : 2,
        f : 4,
    },
}
```

1.3.5 Metrics

class vector_dict.VectorDict.**VectorDict**(**a*, ***a_dict*)

slightly enhanced Dict

norm()

norm of a vector dict = sqrt(a . a)

dot(*other*)

scalar = sum items self * other for each distinct key in common norm of the projection of self on other

jaccard(*other*)

jaccard similariry of two vectors dicts a . b / (||a||^2 + ||b||^2 - a . b)

cos(*other*)

cosine similarity of two vector dicts a . b / (||a||*||b||)

1.3.6 Aliases

class vector_dict.VectorDict.**cos**

for ease of reading and writing equivalent to the cosinus similarity obj1.cos(2) returns the cosinus similarity of two vectorDict

```
>>> cos(
       VectorDict( int, dict( x=1 , y=1) ),
       VectorDict( int, dict( x=1 , y=0) ),
    )
0.7071067811865475
```

class vector_dict.VectorDict.**dot**

for ease of reading and writing equivalent to obj1.dot(obj2) does the leaf by leaf product of the imbricated dict for all the keys that are similar.

```
>>> dot(
       VectorDict( int, dict( x=1 , y=1, z=0) ),
       VectorDict( int, dict( x=1 , y=1, z=1) ),
    )
2.0
```

1.3.7 exemples

Making map reduce in mongodb fashion

```
from vector_dict.VectorDict import iter_object,  VectorDict
from csv import reader, writer
import os
from io import StringIO
from numpy import array

### CSV of langage users in a nosql DB (or any cloudish crap)
### having individual votes of cooolness per langage
### should be a very large dataset to be funnier

mocking_nosql = u"""george,FR,fun,perl,2
roger,FR,serious,python,3
christine,DE,fun,python,3
bob,US,serious,php,1
isabelle,FR,fun,perl,10
Kallista,FR,unfun,c#,-10
Nellev,FR,typorigid,python,1
haypo,FR,javabien,python,1
potrou,FR,globally locked,python,1
petra,DE,sexy,VHDL,69"""

nosql_iterator = reader

### interesting columns/key we want to emit
COUNTRY = 1
LANGAGE = 3
COOLNESS = 4

w = writer(os.sys.stdout)

## basic example for counting langage users nothing more interesting than
## what collections.counter does
print ""
w.writerow(["langage", "how many users"])
map(w.writerow,
    iter_object(
        reduce(
            ## well that is a complex reduce operation :)
            VectorDict.__iadd__,
            map(
                lambda document: VectorDict(int, {document[LANGAGE]: 1}),
                nosql_iterator(StringIO(mocking_nosql))
            )
        ),
        flatten = True
    )
)
"""expected result :
langage,how many users
python,3
c#,1
php,1
VHDL,1
perl,2
```

```
"""
print "\n" * 2 + "next test\n"
### example with group by + aggregation of a counter
### counting all langage per country with their coolness and number of users
### Hum .... I miss a sort and a  limit by to be completely SQL like compatible
w.writerow(["country", "langage", "coolness", "howmany"])
map(w.writerow,
    iter_object(
        ##nosql like reduce
        reduce(
            ## well that is the same very complex reduce operation :)
            VectorDict.__iadd__,
            ##nosql like map where we emit interesting subset of the record
            map(
                lambda document: VectorDict(
                    VectorDict, {
                        #KEY
                        document[COUNTRY]:
                        VectorDict(
                            array,
                            {
                                #GROUPBY
                                document[LANGAGE]:
                                #AGGREGATOR
                                array([
                                    #Counter
                                    int(document[COOLNESS]),
                                    #presence
                                    1
                                ])
                            }
                        ),
                        ## making a (sub) total on the fly
                        'total_coolness_and_voters': array([
                            int(document[COOLNESS]), 1
                        ])
                    }
                ),

                ## nosql like filter that should be in the map if it was nosql
                ## maybe we also need a map that accepts None as a result and
                ## skip the result in the resulting iterator,
                ## or a skip() callable in lambda ?
                ## or sub()  like in perl with curly braces
                ## joke : combining map / filter is easy enough
                filter(
                    lambda document: "php" != document[LANGAGE],
                    nosql_iterator(StringIO(mocking_nosql))
                )
            )
        ),
            flatten = True
    )
)
"""expected result :
country,langage,coolness,howmany
FR,python,4,2
FR,c#,-10,1
```

```
FR,perl,12,2
total_votes_and_coolness,7,78
DE,python,3,1
DE,VHDL,69,1
"""
```

Selecting item in a tree

```
from vector_dict.VectorDict import VectorDict, convert_tree, is_leaf
from vector_dict.Clause import *
def findme( v, a_tree):
    if not is_leaf(v):
        return v.match_tree(a_tree)

positive = lambda v : v > 0

def pretty_present(list):
    print "Result "
    for el in list:
        print "path %r " % el[0]
        print "has value %s" %  (hasattr(el[1], 'tformat') and el[1].tformat() or el[1] )

    print
w = convert_tree( dict(
    a = dict( c=  1),
    e = dict( d= 3.0) ,
    b = dict( c = 1 , d = 4 )
))

w.tprint()

pretty_present( w.find( lambda p, v : has_all( 'c', 'd' )(v) ) )

pretty_present( w.find( lambda p, v : findme(v, dict(
                                        d= has_type(int),
                                        c=has_type(int)   )
) ) )

pretty_present( w.find( lambda p, v : has_tree({  'a' : { 'c' : positive  } })(v)) )

pretty_present( w.find( lambda p, v : (
        has_type( int )(v) or has_type(float)(v)
    ) and  v > 3  ))
pretty_present( w.find( lambda p, v : p.endswith( [ 'c' ] ) ) )


"""
RESULTS :
{
    a = {
        c = 1,
    },
    b = {
        c = 1,
        d = 4,
    },
    e = {
```

```
        d = 3.0,
    },
}
Result
path ['b']
has value {
    c = 1,
    d = 4,
}

Result
path ['b']
has value {
    c = 1,
    d = 4,
}

Result
path []
has value {
    a = {
        c = 1,
    },
    b = {
        c = 1,
        d = 4,
    },
    e = {
        d = 3.0,
    },
}

Result
path ['b', 'd']
has value 4

Result
path ['a', 'c']
has value 1
path ['b', 'c']
has value 1
"""
```

Word counting with multiprocess and vector dict

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
""" parallel wordcounting with VectorDict
Notice we can also get all first letter count in the same operations
unlike regular word count methods"""

## Making the input
# wget http://www.gutenberg.org/cache/epub/26740/pg26740.txt
# mv pg26740.txt dorian.txt
# split -l 2125 dorian.txt dorian_splited.

import os, sys, inspect
```

```
cmd_folder = os.path.abspath(
    os.path.join(
        os.path.split(
            inspect.getfile( inspect.currentframe() )
        )[0] ,
        ".."
    )
)
if cmd_folder not in sys.path:
   sys.path.insert(0, cmd_folder)


from multiprocessing import Pool
import string
import re
#from codecs import open as open

from vector_dict.VectorDict import VectorDict as vd
FILES = [ "dorian_splited.aa",  "dorian_splited.ab",
        "dorian_splited.ac",  "dorian_splited.ad" ]


def word_count( unicode_file ):

    exclude = set(string.punctuation)
    def clean(exlcude):
        def _clean(word):
            return ''.join(ch for ch in word if ch not in exclude)
        return _clean

    sp_pattern = re.compile( """[\.\!\"\s\-\,\']+""", re.M)
    res = vd( int, {})
    for line in iter(open(unicode_file ) ):
        for word in map( clean(exclude),
                map( str.lower, sp_pattern.split(line ))
            ):
            if len(word) > 2 :
                res += vd(int, {
                    word : 1 ,
                    'begin_with' :
                        vd(int, { word[0] : 1 }) ,
                    'has_size' :
                        vd(int, { len(word) : 1 } )
                    })

    return res

p = Pool()
result=p.map(word_count, FILES )
result = reduce(vd.__add__,result)
print "Frequency of words begining with"
result['begin_with'].tprint()
result.prune( "begin_with")
print "Repartition of size words size"
result['has_size'].tprint()
result.prune( "has_size")

from itertools import islice
```

```
print "TOP 40 most used words"
print "\n".join(
     "%10s=%s" % (x, y) for x, y in sorted(result.items(), key=lambda x: x[1],reverse=True)[:40]

)

"""
EXPECTED RESULTS :
Frequency of words begining with
{
    '\xc3' : 4,
    'r' : 1426,
    'a' : 5372,
    '1' : 10,
    'c' : 2778,
    'b' : 2659,
    'e' : 1420,
    'd' : 2564,
    'g' : 1534,
    'f' : 2647,
    'i' : 942,
    'h' : 5867,
    'k' : 483,
    'j' : 245,
    'm' : 2567,
    'l' : 2706,
    'o' : 1669,
    'n' : 1416,
    'q' : 199,
    'p' : 2173,
    's' : 5434,
    '2' : 2,
    'u' : 407,
    't' : 9255,
    'w' : 5491,
    'v' : 507,
    'y' : 2077,
    'x' : 9,
    'z' : 3,
}
TOP 40 most used words
       the=3786
       and=2216
       you=1446
      that=1362
       was=1083
       his=995
       had=833
      with=661
       him=661
       for=588
      have=561
       not=474
       her=439
       she=431
    dorian=420
      what=399
       but=396
```

```
       one=393
       are=372
     there=337
      they=318
     would=308
       all=294
      said=262
       don=255
      from=254
      were=251
      lord=248
     henry=237
      been=236
      life=232
      like=227
       who=224
     about=223
      when=218
      your=207
      some=205
      gray=205
      them=203
      will=201
"""
```

1.4 Accessing, selecting and modifying elements in a tree

For convenience purpose, the result of the find method returns a namedtuple Element where

- Element[0] is the same as Element.path : the path to the value,
- Element[1] is the same as Element.value : the value

1.4.1 predicates on path

As path is of the Path class, you can use *endswith*, *startswith* and *countains* methods.

```
>>> from vector_dict.VectorDict import convert_tree, VectorDict
>>> a = convert_tree( { 'a' : { 'a' : { 'r' : "yop" , 'b' : { 'c' :  1 }, 'd' : True } } } )
>>> a.pprint()
a->a->r = 'yop'
a->a->b->c = 1
a->a->d = True
>>> a.tprint()
{
    a : {
        a : {
            r : 'yop',
            b : {
                c : 1,
            },
            d : True,
        },
    },
}
```

```
>>> print "\n".join([
        "%r => %r"  %( e.path, e.value ) for e in a.find(
            lambda k, v : k.endswith( 'a','r' ) )
    ])
 ('a', 'a', 'r') => 'yop'
```

If you ask a condition on path only, it will return **all** nodes verifying the condition

code continued from first example

```
>>> a.build_path( "a", "a", "g", "a", 2 )
>>> [ e.path for e in a.find( lambda k,v : k.contains( 'a' ) ) ]
[('a',), ('a', 'a'), ('a', 'a', 'r'), ('a', 'a', 'b'), ('a', 'a', 'b', 'c'), ('a', 'a', 'd'), ('a', '
```

1.4.2 Limiting the matching elements to leaves

If you dont want any answers returned in the form of a tree add the is_leaf clause :

code continued from first example

```
>>> from vector_dict.Clause import is_leaf
>>> [ e.path for e in a.find( lambda k,v : k.contains( 'a' )  and is_leaf(v)) ]
[('a', 'a', 'r'), ('a', 'a', 'b', 'c'), ('a', 'a', 'd'), ('a', 'a', 'g', 'a')]
>>> [ e.value for e in a.find( lambda k,v : k.contains( 'a' )  and is_leaf(v)) ]
['yop', 1, True, 2]
>>> [ e for e in a.find( lambda k,v : k.contains( 'a' )  and is_leaf(v)) ]
[Element(path=('a', 'a', 'r'), value='yop'), Element(path=('a', 'a', 'b', 'c'), value=1), Element(pat
>>> a.tprint()
{
    a : {
        a : {
            r : 'yop',
            b : {
                c : 1,
            },
            d : True,
            g : {
                a : 2,
            },
        },
    },
}
```

1.4.3 Searching a specified location of a tree

You can also narrow your search on a subtree with the given path by combining with at() method. For instance searching in 'a' and 'b' subtree

```
>>> from vector_dict.VectorDict import convert_tree, VectorDict,Element,Path
>>> a = convert_tree(dict(a=dict(x=1, y=2, z=3), b=dict( x=-1, y=-1, z = -2)))
>>> [ e for e in a.at( 'b' ).find( lambda k, v : k.endswith('x')) ]
[Element(path=('x',), value=-1)]
>>> [ e for e in a.at( 'a' ).find( lambda k, v : k.endswith('x')) ]
[Element(path=('x',), value=1)]
```

1.4.4 Mainpulating values yielded by the find method

Also with at you can manipulate values :

code continued from the above example

```
>>> [ a.at(e.path , lambda v : v * -10 ) for e in a.find( lambda k, v : k.endswith('x')) ]
[-10, 10]
>>> a.tprint()
{
    a : {
        y : 2,
        x : -10,
        z : 3,
    },
    b : {
        y : -1,
        x : 10,
        z : -2,
    },
}
```

Warning:

- Since it is pretty not a good idea to change a collection while it is being iterated any in situ search / replace at the same time is strongly discouraged;
- Always work with the path when manipulating

1.5 How to use addition, substraction, division and multiplication

Pretty sanely it is a leaf by leaf operation, and will work as long as element supports the operations.

```
>>> from vector_dict.VectorDict import convert_tree, VectorDict,Element,Path
>>> a = convert_tree(dict(a = dict(aint=1,afloat=2.0  , anarray=[1,2], astring="yo"), c= False))
>>> (a+a).tprint()
{
    a = {
        aint : 2,
        anarray : [1, 2, 1, 2],
        astring : 'yoyo',
        afloat : 4.0,
    },
    c = 0,
}
>>> (a*2).tprint()
{
    a : {
        aint : 2,
        anarray : [1, 2, 1, 2],
        astring : 'yoyo',
        afloat : 4.0,
    },
    c : 0,
}
>>> (a*a).tprint()
#Traceback (most recent call last):
#File "<input>", line 1, in <module>
```

```
#File "vector_dict/VectorDict.py", line 537, in __mul__
#  return self.__opfactory__(other, True)
#File "vector_dict/VectorDict.py", line 557, in __opfactory__
#  return getattr(a_copy, intern_operation)(other)
#File "vector_dict/VectorDict.py", line 852, in __internal_mul__
#  new_dict[k] = (self[k]).__internal_mul__( other[k] )
#File "vector_dict/VectorDict.py", line 854, in __internal_mul__
#  new_dict[k] = self[k] * other[k]
#TypeError: can't multiply sequence by non-int of type 'list'
```

It also works (in the rules of conservation) by creating new path/value in the new tree if it does not exists for addition. In multiplication if values are not present in both trees they are skipped.

```
>>> a = convert_tree(dict(a = dict(x=1, y=2.0, z=3), c= 1))
>>> b = convert_tree(dict(a = dict(x=-1.0, y=2.0, z=6), d= 1))
>>> (a+b).tprint()
{
    a : {
        y : 4.0,
        x : 0.0,
        z : 9,
    },
    c : 1,
    d : 1,
}
>>> (a*b).tprint()
{
    a : {
        y : 4.0,
        x : -1.0,
        z : 18,
    },
}
>>> (a-b).tprint()
{
    a : {
        y : 0.0,
        x : 2.0,
        z : -3,
    },
    c : 1,
    d : -1,
}
>>> (1.0*a/b).tprint()
{
    a : {
        y : 1.0,
        x : -1.0,
        z : 0.5,
    },
}
```

1.6 Sparse Matrix

Dict behaving like a vector, and supporting all operations the algebraic way

1.6.1 API

class vector_dict.SparseMatrix.**SparseMatrix**(**coord_fonc*, ***kw*)

Sparse Matrice on dict of dict of dict a sparse matrix is a set of

- src path coordinate
- dst path coordinate
- and a fonction to apply on the src value

to transform it in destination value

1.6.2 Example

```
from vector_dict.Clause import Clause, is_leaf, is_container

from vector_dict.Operation import  identity, mul, cast
from vector_dict.VectorDict import convert_tree
from vector_dict.SparseMatrix import SparseMatrix,Coordinates
from collections import namedtuple, defaultdict

a = convert_tree( { 'a' : { 'b' : 1 , 'c' : 2 } , 'b' : 0 } )

def title(string):
    print
    print "*" * 80
    print string
    print "*" * 80
title( "initial dictionary" )
a.tprint()

m = SparseMatrix(
    ## take source['a']['b'] and * -2 and put it in dst['mul']['neg2']
    ( tuple( [ 'a', 'b' ] ), tuple([ 'mul', 'neg2' ] ), mul(-2) ),
    ## guess :)
    ( tuple([ 'a', 'c' ]), tuple([ 'mul', 'misplaced' ]), cast(float) ),
    ( tuple([ 'b' ])  , tuple([ 'a' ]) , lambda x : -4 ),
    ( tuple( [ 'a' ] ), tuple( [ 'a_dict' ] ), identity ),
    )
title( "transformed dict" )
m(a).tprint()
w = SparseMatrix()
w[
    Coordinates( src= tuple( [ ] ) ,
    dst =  tuple([ 'a_copy' ] ))
]=  identity
w[
    Coordinates( src= tuple( [ ] ) ,
    dst =  tuple([ 'inception' ] ))
]=  m.copy()

title( "with matrix in matrix" )
w(a).tprint()

"""
********************************************************************************
initial dictionary
```

```
********************************************************************************
{
    a = {
        c = 2,
        b = 1,
    },
    b = 0,
}

********************************************************************************
transformed dict
********************************************************************************
{
    a = -4,
    mul = {
        neg2 = -2,
    },
    a_dict = {
        c = 2,
        b = 1,
    },
}

********************************************************************************
with matrix in matrix
********************************************************************************
{
    inception = {
        a = -4,
        mul = {
            neg2 = -2,
        },
        a_dict = {
            c = 2,
            b = 1,
        },
    },
    a_copy = {
        a = {
            c = 2,
            b = 1,
        },
        b = 0,
    },
}
"""
```

1.7 Clause and operations

1.7.1 Clause

Clauses are mainly used for with VectorDict.find Clauses for common search on tree

`vector_dict.Clause.`**`anything`** **= <vector_dict.Clause.Clause object at 0x21977d0>**
matches anything

vector_dict.Clause.**find_re** = <vector_dict.Clause.Clause object at 0x2197510>
Find if something match a regular expression

vector_dict.Clause.**has_tree** = <vector_dict.Clause.Clause object at 0x2197790>
Check if value has the subtree included in it

vector_dict.Clause.**has_type** = <vector_dict.Clause.Clause object at 0x21975d0>
Check if value has type _type

vector_dict.Clause.**is_container** = <vector_dict.Clause.Clause object at 0x2197550>
to know if something can contain more than one value (list , iterator, ...)

vector_dict.Clause.**is_function**(*v*)
check if value is a function

vector_dict.Clause.**is_leaf** = <vector_dict.Clause.Clause object at 0x2197590>
is_leaf if value is not instance of dict

1.7.2 Operation

Operation are mainly used for with VectorDict.at and find and Sparse Matrix misc function that can be used on tree or leaf just for convenience

identity [] return itself

copy [] return a copy of a value

mul(by) [] return a function that multiples by a value

cast(type) [] cast the element in given type

1.8 Path

Enhancement on tuple logical operations to ease the search on tuple of keys.

1.8.1 API

class vector_dict.VectorDict.**Path**(*a_tuple*)

contains(**a_tuple*)
checks if the serie of keys is contained in a path

```
>>> p = Path( [ 'a', 'b', 'c', 'd' ] )
>>> p.contains( 'b', 'c' )
>>> True
```

endswith(**a_tuple*)
check if path ends with the consecutive given has argumenbts value

```
>>> p = Path( [ 'a', 'b', 'c' ] )
>>> p.endswith( 'b', 'c' )
>>> True
>>> p.endswith( 'c', 'b' )
>>> False
```

startswith(**a_tuple*)
checks if a path starts with the value

```
>>> p = Path( [ 'a', 'b', 'c', 'd' ] )
>>> p.startswith( 'a', 'b' )
>>> True
```

1.9 ConsistentAlgebrae: testing algebrae consistency

This class test for any object conformance with linear algebrae rules

class vector_dict.ConsistentAlgebrae.**ConsistentAlgebrae**(***kw*)
test wether an addition for two object is consistant

__init__(***kw*)
only method really callable. Arguments :

- •neutral : neutral element of addition for the object ;
- •scalar : real, float, or complex (normaly anything that is 1D, and follow algebraic rules);
- •one : an element to test
- •other : other element to test

optionnal :

- other_scalar : real, float, or complex (normaly anything that is 1D, and follow algebraic rules);
- context : default "print" make it verbose ;
- collect_values : default lambda x : x, if testing for conservation a lambda fonction for getting the values

1.9.1 How to use it

```
if cmd_folder not in sys.path:
   sys.path.insert(0, cmd_folder)

try:
    from numpy import array as array

    ConsistentAlgebrae(
        neutral=array([0, 0, 0]),
        one=array([1, 2, 3]),
        another=array([5, 2, 3]),
        other=array([3, 4, -1]),
        equal=lambda left, right: (right == left).all(),
        )
except Exception as e:
    print "only lamers dont use numpy"


ConsistentAlgebrae(
    neutral=0,
    one=1,
    other=2,
    another=3
```

```
    )


ConsistentAlgebrae(
    neutral=[],
    one=[1],
    other=[2],
    another=[42]
    )

ConsistentAlgebrae(
    neutral="",
    one="1",
    other="2",
    another="4"
    )

ConsistentAlgebrae(
    neutral=VectorDict(int, {}),
    one=VectorDict(int, {"one": 1, "one_and_two": 3}),
    other=VectorDict(int, {"one_and_two": - 1, "two": 2}),
    another=VectorDict(int, {"one": 3, 'two':  2, "three": 1}),
    collect_values=lambda x: x.values()
    )

one = VectorDict(int, {"one": 1, "one_and_two": 12})
other = VectorDict(int, {"one_and_two": - 9, "two": 2})

print "just for fun \n\t%r\n\t+\n\t%r\n\t=\n\t%r" % (one, other, one + other)
```

1.9.2 expected results

1.10 Convention :

version x.y.z

while in beta convention is :

- **x** = 0
- **y** = API change
- **z** = bugfix and/or improvement

and then

- **x** = API change
- **y** = improvement
- **z** = bugfix

1.11 Changelog

v 0.3.0 VectorDict adding some easy features matrix, vector basic operation (get, at, prune, find)

v 0.3.1 Bugfix on build_path Adding tests

v 0.4.0 inlining example with methods so that it is more obvious what method does

- removing add_path can be replaced by vectordict += tree_from_path(* path)
- more doc
- broke diff

v 0.4.1

- adding doc for finding item in a tree ;
- adding QA information (such as where is the ticketing) in the documentation.

v 0.4.2

- BUG 1 to 6 squashed multiplication/division dont behave correctly
- intersection, union, symmetric difference added in alpha version

v 0.5.0

- BUG 7 is not a bug
- removing push
- adding __and__, __or__, __not__
- set operations are Beta, don't use them
- refactoring + - * /
- doc refactoring

v 0.6.0

- BUG 8 fixed
- convert_tree fixed
- pprint fixed
- API __xor__ added
- pformat added
- tformat/tprint output changed I consider this an API change (thus 0.6.0)
- inconsistency in logical operations spotted. Change in API planned (0.7.0)
- how the heck do I force test_suite to be triggered when people install this package and an automated bug tracking trace added to github, plus not installing broken packages ? Python is still way behind perl -eshell -MCPAN behaviour.

v 1.0.0

- BUG 12 fixed
- XOR bug remaining

CHAPTER

TWO

INDICES AND TABLES

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

PYTHON MODULE INDEX

V