Multidimensional array in Python

Asked 15 years, 10 months ago Modified 10 months ago Viewed 27k times

11

I have a little Java problem I want to translate to Python. Therefor I need a multidimensional array. In Java it looks like:

```
double dArray[][][] = new double[x.length()+1][y.length()+1]
[x.length()+y.length()+3];
dArray[0][0][0] = 0;
dArray[0][0][1] = POSITIVE_INFINITY;
```

Further values will be created bei loops and written into the array.

How do I instantiate the array?

PS: There is no matrix multiplication involved...

python arrays

Share
Improve this question
Follow

edited Feb 22 at 21:11

mkrieger1
22.8k ● 7 ● 63 ● 79

asked Feb 3, 2009 at 19:54

Christian Stade-Schuldt
4,841 ● 7 ● 37 ● 31

Incidentally, you can get floating point infinity in python as `float('inf')`. It behaves more-or-less as you would expect – most operations just give you inf, and a few give you nan. – John Fouhy Feb 4, 2009 at 0:32

(merged with an earlier question; kept this as the master as a: the OP still exists, b: the question is better-phrased, and c: to avoid lots of necromancers) – Marc Gravell Nov 26, 2009 at 12:22

12 Answers

Sorted by: Highest score (default)

20

If you restrict yourself to the Python standard library, then a list of lists is the closest construct:

```
arr = [[1,2],[3,4]]
```

gives a 2d-like array. The rows can be accessed as `arr[i]` for `i` in `{0,..,len(arr}`, but column access is difficult.

If you are willing to add a library dependency, the NumPy package is what you really want. You can create a fixed-length array from a list of lists using:

```
import numpy
arr = numpy.array([[1,2],[3,4]])
```

Column access is the same as for the list-of-lists, but column access is easy: `arr[:,i]` for `i` in `{0,..,arr.shape[1]}` (the number of columns).

In fact NumPy arrays can be n-dimensional.

Empty arrays can be created with

```
numpy.empty(shape)
```

where `shape` is a tuple of size in each dimension; `shape=(1,3,2)` gives a 3-d array with size 1 in the first dimension, size 3 in the second dimension and 2 in the 3rd dimension.

If you want to store objects in a NumPy array, you can do that as well:

```
arr = numpy.empty((1,), dtype=numpy.object)
arr[0] = 'abc'
```

For more info on the NumPy project, check out the NumPy homepage.

Share

Improve this answer

Follow

edited Dec 9, 2022 at 19:01

Glorfindel

22.6k ● 13 ● 89 ● 116

answered Nov 4, 2008 at 6:37

Barry Wark

108k ● 24 ● 182 ● 206

There is no need to use tuple. `numpy.empty([1])` works as well. – jfs Sep 29, 2009 at 19:53

The simple "random" number generator: `noise = lambda: numpy.empty([1])[0]` – jfs Sep 29, 2009 at 20:00

True, any iterable (list, tuple, etc.) works as the dimensions argument. Your "random" number generator relies on the uninitialized memory being "random" and should *never* be used as a source of randomness for anything that maters (e.g. cryptography). – Barry Wark Sep 29, 2009 at 20:18

@Barry: True, that's why I've put quotes around `random` – jfs Oct 1, 2009 at 1:48

@JFSebastian: I'm sure you know the difference. Just wanted to make sure the Google did too, for posterity and all that. – Barry Wark Oct 1, 2009 at 16:24

15

To create a standard python array of arrays of arbitrary size:

```
a = [[0]*cols for _ in [0]*rows]
```

It is accessed like this:

```
a[0][1] = 5 # set cell at row 0, col 1 to 5
```

A small python gotcha that's worth mentioning: It is tempting to just type

```
a = [[0]*cols]*rows
```

but that'll copy the *same* column array to each row, resulting in unwanted behaviour. Namely:

```
>>> a[0][0] = 5
>>> print a[1][0]
5
```

Share

Improve this answer

Follow

edited Nov 4, 2008 at 9:40

answered Nov 4, 2008 at 9:34

Deestan

17.1k ● 4 ● 34 ● 48

4 I'd rather use: `[[0]*cols for _ in xrange(rows)]` – jfs Sep 29, 2009 at 19:48

@J.F.Sebastian Ya me too, really. I just found the symmetry appealing at the time of writing this. :-) – Deestan Feb 27, 2012 at 10:24

plus1 for the gotcha. – AShelly Apr 7, 2015 at 1:53

13

You can create it using nested lists:

```
matrix = [[a,b],[c,d],[e,f]]
```

If it has to be dynamic it's more complicated, why not write a small class yourself?

```
class Matrix(object):
    def __init__(self, rows, columns, default=0):
        self.m = []
        for i in range(rows):
            self.m.append([default for j in range(columns)])
```

```
    def __getitem__(self, index):
        return self.m[index]
```

This can be used like this:

```
m = Matrix(10,5)
m[3][6] = 7
print m[3][6] // -> 7
```

I'm sure one could implement it much more efficient. :)

If you need multidimensional arrays you can either create an array and calculate the offset or you'd use arrays in arrays in arrays, which can be pretty bad for memory. (Could be faster though…) I've implemented the first idea like this:

```
class Matrix(object):
    def __init__(self, *dims):
        self._shortcuts = [i for i in self._create_shortcuts(dims)]
        self._li = [None] * (self._shortcuts.pop())
        self._shortcuts.reverse()

    def _create_shortcuts(self, dims):
        dimList = list(dims)
        dimList.reverse()
        number = 1
        yield 1
        for i in dimList:
            number *= i
            yield number

    def _flat_index(self, index):
        if len(index) != len(self._shortcuts):
            raise TypeError()

        flatIndex = 0
        for i, num in enumerate(index):
            flatIndex += num * self._shortcuts[i]
        return flatIndex

    def __getitem__(self, index):
        return self._li[self._flat_index(index)]

    def __setitem__(self, index, value):
        self._li[self._flat_index(index)] = value
```

Can be used like this:

```
m = Matrix(4,5,2,6)
m[2,3,1,3] = 'x'
m[2,3,1,3] // -> 'x'
```

Improve this answer

Follow

Georg Schölly
126k • 54 • 223 • 276

11

Take a look at numpy

here's a code snippet for you

```
import numpy as npy

d = npy.zeros((len(x)+1, len(y)+1, len(x)+len(y)+3))
d[0][0][0] = 0 # although this is unnecessary since zeros initialises to zero
d[i][j][k] = npy.inf
```

I don't think you need to be implementing a scientific application to justify the use of numpy. It is faster and more flexible and you can store pretty much anything. Given that I think it is probably better to try and justify *not* using it. There are legitimate reasons, but it adds a great deal and costs very little so it deserves consideration.

P.S. Are your array lengths right? It looks like a pretty peculiar shaped matrix...

Share

Improve this answer

Follow

edited Dec 9, 2022 at 19:01

Glorfindel
22.6k • 13 • 89 • 116

answered Feb 3, 2009 at 20:00

Simon
80.6k • 26 • 92 • 119

10

Multidimensional arrays are a little murky. There are few reasons for using them and many reasons for thinking twice and using something else that more properly reflects what you're doing. [Hint. your question was thin on context ;-)]

If you're doing matrix math, then use `numpy`.

However, some folks have worked with languages that force them to use multi-dimensional arrays because it's all they've got. If your as old as I am (I started programming in the 70's) then you may remember the days when multidimensional arrays were the only data structure you had. Or, your experience may have limited you to languages where you had to morph your problem into multi-dimensional arrays.

Say you have a collection *n* 3D points. Each point has an x, y, z, and time value. Is this an *n* x 4 array? Or a 4 * *n* array? Not really.

Since each point has 4 fixed values, this is more properly a list of tuples.

```
a = [ ( x, y, z, t ), ( x, y, z, t ), ... ]
```

Better still, we could represent this as a list of objects.

```
class Point( object ):
    def __init__( self, x, y, z, t ):
        self.x, self.y, self.z, self.t = x, y, z, t

a = [ Point(x,y,x,t), Point(x,y,z,t), ... ]
```

Share Improve this answer Follow

answered Nov 4, 2008 at 12:04

S.Lott

391k ● 82 ● 517 ● 788

5

If you are OK using sparse arrays, you could use a dict to store your values. Python's dicts allow you to use tuples as keys, as such, you could assign to and access elements of the "sparse array" (which is really a dict here) like this:

```
d = {}
d[0,2,7] = 123 # assign 123 to x=0, y=2, z=7
v = d[0,2,7]
```

Share Improve this answer Follow

answered Feb 3, 2009 at 20:04

paprika

2,484 ● 26 ● 46

Nice trick to store a multidimensional array. However things would get very messy when trying to iterate over elements say of a column, or a row, or even to be able to get the 'dimensions'. – Sergey Golovchenko Feb 3, 2009 at 20:40

Indeed. If I were to use something like this, I'd of course encapsulate this into a class and introduce methods for read/write access. An instance variable would store the dimension of the array and the read/write methods would return a default value for unassigned elements (by catching KeyError). – paprika Feb 3, 2009 at 21:49

+1: dictionary is considerably simpler. Iteration over dimensions is easy, not messy. xRange = set([k[0] for k in d.keys()]), yRange = set([k[1] for k in d.keys()]) – S.Lott Feb 3, 2009 at 22:46

@S.Lott: what I mean by messy is that iteration that you suggest would only work if you have filled all the elements of this multidimensional array, however if you have "gaps" this simply wouldn't work. anyways +1 to paprika for an original solution. – Sergey Golovchenko Feb 4, 2009 at 4:43

4

Probably not relevant for you but if you are doing serious matrix work see numpy

Share

Improve this answer

Follow

edited Dec 9, 2022 at 19:01

Glorfindel

22.6k ● 13 ● 89 ● 116

answered Feb 3, 2009 at 20:00

Martin Beckett

96k ● 28 ● 195 ● 268

3

For numeric data, Numpy Arrays:

```
>>> matrix1 = array(([0,1],[1,3]))
>>> print matrix1
[[0 1]
[1 3]]
```

For general data (e.g. strings), you can use a list of lists, list of tuples, ...

```
matrix2 = [['a','b'], ['x','y']]
```

Share

Improve this answer

Follow

edited Dec 9, 2022 at 19:01

Glorfindel

22.6k ● 13 ● 89 ● 116

answered Nov 4, 2008 at 5:47

gimel

86.1k ● 10 ● 79 ● 104

Can I initialize it a given number of rows and columns like in C? – doekman Nov 4, 2008 at 5:54

Mutable objects (e.g. list) can change - so initialization is not always relevant. In numpy, an Array has a Shape (dimensions) that can be modified without changing the data. – gimel Nov 4, 2008 at 6:02

1 @Popper: Yes. zeros((3,4)) will create a 3x4 array of zeros. – S.Lott Nov 4, 2008 at 12:16

2

Here's a quick way to create a nested 3-dimensional list initialized with zeros:

```
# dim1, dim2, dim3 are the dimensions of the array
a =[[[0 for _ in range(dim1)] for _ in range(dim2)] for _ in range(dim1) ]
a[0][0][0] = 1
```

this is a list of lists of lists, a bit more flexible than an array, you can do:

```
a[0][0] = [1,2,3,4]
```

to replace a whole row in the array, or even abuse it like that:

```
a[0] = "Ouch"
print a[0][0] #will print "O", since strings are indexable the same way as
```

```
lists
print a[0][0][0] #will raise an error, since "O" isn't indexable
```

but if you need performance, then I agree that numpy is the way to go.

Also, beware of:

```
a = [[[0] * 5]*5]*5]
```

If you try `a[0][0][0]=7` on the object above, you will see what's wrong with that.

Share Improve this answer Follow

answered Feb 3, 2009 at 21:34

Rafał Dowgird
45k • 11 • 79 • 94

0

Another option is to use a dictionary:

```
>>> from collections import defaultdict
>>> array = defaultdict(int) # replace int with the default-factory you want
>>> array[(0,0)]
0
>>> array[(99,99)]
0
```

You'll need to keep track of the upper & lower bounds as well.

Share Improve this answer Follow

answered Nov 5, 2008 at 7:33

Matthew Trevor
14.9k • 6 • 40 • 51

0

I've just stepped into a similar need and coded this:

```
def nDimensionsMatrix(dims, elem_count, ptr=[]):
    if (dims > 1):
        for i in range(elem_count[dims-1]):
            empty = []
            ptr.append(empty)
            nDimensionsMatrix(dims-1, elem_count, empty)
        return ptr
    elif dims == 1:
        ptr.extend([0 for i in range(elem_count[dims])])
        return ptr

matrix = nDimensionsMatrix(3, (2,2,2))
```

I'm not looking at speed, only funcionality ;)

I want to create a matrix with N dimensions and initialize with 0 (a *elem_count* number of elements in each dimension).

Hope its helps someone

Share Improve this answer Follow

answered Jul 28, 2009 at 15:31

fabiopedrosa

2,581 ● 8 ● 29 ● 42

0

Easy, when using numpy:

```
b = ones((2,3,4)) # creates a 2x3x4 array containing all ones.
```

'ones' can be replaced with 'zeros'

Share

Improve this answer

Follow

edited Sep 30, 2009 at 4:27

sth

229k ● 56 ● 286 ● 368

answered Sep 29, 2009 at 19:29

HenryJP