NumPy - Iterating Over Array

NumPy package contains an iterator object **numpy.nditer**. It is an efficient multidimensional iterator object using which it is possible to iterate over an array. Each element of an array is visited using Python's standard Iterator interface.

Let us create a 3X4 array using arange() function and iterate over it using **nditer**.

Example 1

Live Demo

```
import numpy as np

a = np.arange(0,60,5)

a = a.reshape(3,4)



print 'Original array is:'

print a

print '\n'



print 'Modified array is:'

for x in np.nditer(a):

   print x,
```

The output of this program is as follows −

```
Original array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Modified array is:
0 5 10 15 20 25 30 35 40 45 50 55
```

Example 2

The order of iteration is chosen to match the memory layout of an array, without considering a particular ordering. This can be seen by iterating over the transpose of the above array.

Live Demo

```
import numpy as np

a = np.arange(0,60,5)

a = a.reshape(3,4)


print 'Original array is:'

print a

print '\n'


print 'Transpose of the original array is:'

b = a.T

print b

print '\n'


print 'Modified array is:'

for x in np.nditer(b):

   print x,
```

The output of the above program is as follows −

```
Original array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Transpose of the original array is:
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]

Modified array is:
```

```
0 5 10 15 20 25 30 35 40 45 50 55
```

Iteration Order

If the same elements are stored using F-style order, the iterator chooses the more efficient way of iterating over an array.

Example 1

Live Demo

```
import numpy as np

a = np.arange(0,60,5)

a = a.reshape(3,4)

print 'Original array is:'

print a

print '\n'


print 'Transpose of the original array is:'

b = a.T

print b

print '\n'


print 'Sorted in C-style order:'

c = b.copy(order = 'C')

print c

for x in np.nditer(c):

   print x,


print '\n'


print 'Sorted in F-style order:'
```

```
c = b.copy(order = 'F')

print c

for x in np.nditer(c):

   print x,
```

Its output would be as follows −

```
Original array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Transpose of the original array is:
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]

Sorted in C-style order:
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]
0 20 40 5 25 45 10 30 50 15 35 55

Sorted in F-style order:
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]
0 5 10 15 20 25 30 35 40 45 50 55
```

Example 2

It is possible to force **nditer** object to use a specific order by explicitly mentioning it.

Live Demo

```
import numpy as np

a = np.arange(0,60,5)

a = a.reshape(3,4)



print 'Original array is:'

print a

print '\n'
print 'Sorted in C-style order:'

for x in np.nditer(a, order = 'C'):

   print x,

print '\n'


print 'Sorted in F-style order:'

for x in np.nditer(a, order = 'F'):

   print x,
```

Its output would be −

```
Original array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Sorted in C-style order:
0 5 10 15 20 25 30 35 40 45 50 55

Sorted in F-style order:
0 20 40 5 25 45 10 30 50 15 35 55
```

Modifying Array Values

The **nditer** object has another optional parameter called **op_flags**. Its default value is read-only, but can be set to read-write or write-only mode. This will enable modifying array elements using this iterator.

Example

Live Demo

```
import numpy as np

a = np.arange(0,60,5)

a = a.reshape(3,4)

print 'Original array is:'

print a

print '\n'
```

```
for x in np.nditer(a, op_flags = ['readwrite']):

   x[...] = 2*x

print 'Modified array is:'

print a
```

Its output is as follows −

```
Original array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Modified array is:
[[ 0 10 20 30]
 [ 40 50 60 70]
 [ 80 90 100 110]]
```

External Loop

The nditer class constructor has a **'flags'** parameter, which can take the following values −

Sr.No.	Parameter & Description
1	**c_index** C_order index can be tracked
2	**f_index** Fortran_order index is tracked
3	**multi-index** Type of indexes with one per iteration can be tracked
4	**external_loop** Causes values given to be one-dimensional arrays with multiple values instead of zero-dimensional array

Example

In the following example, one-dimensional arrays corresponding to each column is traversed by the iterator.

Live Demo

```
import numpy as np

a = np.arange(0,60,5)

a = a.reshape(3,4)


print 'Original array is:'

print a

print '\n'


print 'Modified array is:'

for x in np.nditer(a, flags = ['external_loop'], order = 'F'):

   print x,
```

The output is as follows −

```
Original array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Modified array is:
[ 0 20 40] [ 5 25 45] [10 30 50] [15 35 55]
```

Broadcasting Iteration

If two arrays are **broadcastable**, a combined **nditer** object is able to iterate upon them concurrently. Assuming that an array **a** has dimension 3X4, and there is another array **b** of dimension 1X4, the iterator of following type is used (array **b** is broadcast to size of **a**).

Example

Live Demo

```
import numpy as np
```

```
a = np.arange(0,60,5)

a = a.reshape(3,4)


print 'First array is:'

print a

print '\n'


print 'Second array is:'

b = np.array([1, 2, 3, 4], dtype = int)

print b

print '\n'


print 'Modified array is:'

for x,y in np.nditer([a,b]):

   print "%d:%d" % (x,y),
```

Its output would be as follows −

```
First array is:
[[ 0 5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Second array is:
[1 2 3 4]

Modified array is:
0:1 5:2 10:3 15:4 20:1 25:2 30:3 35:4 40:1 45:2 50:3 55:4
```