Iterating Over Arrays

The iterator object `nditer` (generated/numpy.nditer.html#numpy.nditer), introduced in NumPy 1.6, provides many flexible ways to visit all the elements of one or more arrays in a systematic fashion. This page introduces some basic ways to use the object for computations on arrays in Python, then concludes with how one can accelerate the inner loop in Cython. Since the Python exposure of `nditer` (generated/numpy.nditer.html#numpy.nditer) is a relatively straightforward mapping of the C array iterator API, these ideas will also provide help working with array iteration from C or C++.

Single Array Iteration

The most basic task that can be done with the `nditer` (generated/numpy.nditer.html#numpy.nditer) is to visit every element of an array. Each element is provided one by one using the standard Python iterator interface.

Example:

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a):
...     print x,
...
0 1 2 3 4 5
```

An important thing to be aware of for this iteration is that the order is chosen to match the memory layout of the array instead of using a standard C or Fortran ordering. This is done for access efficiency, reflecting the idea that by default one simply wants to visit each element without concern for a particular ordering. We can see this by iterating over the transpose of our previous array, compared to taking a copy of that transpose in C order.

Example:

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a.T):
...     print x,
...
0 1 2 3 4 5


>>> for x in np.nditer(a.T.copy(order='C')):
...     print x,
...
0 3 1 4 2 5
```

The elements of both *a* and *a.T* get traversed in the same order, namely the order they are stored in memory, whereas the elements of *a.T.copy(order='C')* get visited in a different order because they have been put into a different memory layout.

Controlling Iteration Order

There are times when it is important to visit the elements of an array in a specific order, irrespective of the layout of the elements in memory. The `nditer` (generated/numpy.nditer.html#numpy.nditer) object provides an *order* parameter to control this aspect of iteration. The default, having the behavior described above, is order='K' to keep the existing order. This can be overridden with order='C' for C order and order='F' for Fortran order.

Example:

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a, order='F'):
...     print x,
...
0 3 1 4 2 5
>>> for x in np.nditer(a.T, order='C'):
...     print x,
...
0 3 1 4 2 5
```

Modifying Array Values

By default, the `nditer` (generated/numpy.nditer.html#numpy.nditer) treats the input array as a read-only object. To modify the array elements, you must specify either read-write or write-only mode. This is controlled with per-operand flags.

Regular assignment in Python simply changes a reference in the local or global variable dictionary instead of modifying an existing variable in place. This means that simply assigning to *x* will not place the value into the element of the array, but rather switch *x* from being an array element reference to being a reference to the value you assigned. To actually modify the element of the array, *x* should be indexed with the ellipsis.

Example:

```
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> for x in np.nditer(a, op_flags=['readwrite']):
...     x[...] = 2 * x
...
>>> a
array([[ 0,  2,  4],
       [ 6,  8, 10]])
```

Using an External Loop

In all the examples so far, the elements of *a* are provided by the iterator one at a time, because all the looping logic is internal to the iterator. While this is simple and convenient, it is not very efficient. A better approach is to move the one-dimensional innermost loop into your code, external to the iterator. This way, NumPy's vectorized operations can be used on larger chunks of the elements being visited.

The `nditer` (generated/numpy.nditer.html#numpy.nditer) will try to provide chunks that are as large as possible to the inner loop. By forcing 'C' and 'F' order, we get different external loop sizes. This mode is enabled by specifying an iterator flag.

Observe that with the default of keeping native memory order, the iterator is able to provide a single one-dimensional chunk, whereas when forcing Fortran order, it has to provide three chunks of two elements each.

Example:

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a, flags=['external_loop']):
...     print x,
...
[0 1 2 3 4 5]


>>> for x in np.nditer(a, flags=['external_loop'], order='F'):
...     print x,
...
[0 3] [1 4] [2 5]
```

Tracking an Index or Multi-Index

During iteration, you may want to use the index of the current element in a computation. For example, you may want to visit the elements of an array in memory order, but use a C-order, Fortran-order, or multidimensional index to look up values in a different array.

The Python iterator protocol doesn't have a natural way to query these additional values from the iterator, so we introduce an alternate syntax for iterating with an `nditer` (generated/numpy.nditer.html#numpy.nditer). This syntax explicitly works with the iterator object itself, so its properties are readily accessible during iteration. With this looping construct, the current value is accessible by indexing into the iterator, and the index being tracked is the property *index* or *multi_index* depending on what was requested.

The Python interactive interpreter unfortunately prints out the values of expressions inside the while loop during each iteration of the loop. We have modified the output in the examples using this looping construct in order to be more readable.

Example:

```
>>> a = np.arange(6).reshape(2,3)
>>> it = np.nditer(a, flags=['f_index'])
>>> while not it.finished:
...     print "%d <%d>" % (it[0], it.index),
...     it.iternext()
...
0 <0> 1 <2> 2 <4> 3 <1> 4 <3> 5 <5>
```

```
>>> it = np.nditer(a, flags=['multi_index'])
>>> while not it.finished:
...     print "%d <%s>" % (it[0], it.multi_index),
...     it.iternext()
...
0 <(0, 0)> 1 <(0, 1)> 2 <(0, 2)> 3 <(1, 0)> 4 <(1, 1)> 5 <(1, 2)>


>>> it = np.nditer(a, flags=['multi_index'], op_flags=['writeonly'])
>>> while not it.finished:
...     it[0] = it.multi_index[1] - it.multi_index[0]
...     it.iternext()
...
>>> a
array([[ 0,  1,  2],
       [-1,  0,  1]])
```

Tracking an index or multi-index is incompatible with using an external loop, because it requires a different index value per element. If you try to combine these flags, the `nditer` (generated/numpy.nditer.html#numpy.nditer) object will raise an exception

Example:

```
>>> a = np.zeros((2,3))
>>> it = np.nditer(a, flags=['c_index', 'external_loop'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Iterator flag EXTERNAL_LOOP cannot be used if an index or
multi-index is being tracked
```

Buffering the Array Elements

When forcing an iteration order, we observed that the external loop option may provide the elements in smaller chunks because the elements can't be visited in the appropriate order with a constant stride. When writing C code, this is generally fine, however in pure Python code this can cause a significant reduction in performance.

By enabling buffering mode, the chunks provided by the iterator to the inner loop can be made larger, significantly reducing the overhead of the Python interpreter. In the example forcing Fortran iteration order, the inner loop gets to see all the elements in one go when buffering is enabled.

Example:

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a, flags=['external_loop'], order='F'):
...     print x,
...
[0 3] [1 4] [2 5]

>>> for x in np.nditer(a, flags=['external_loop','buffered'], order='F
'):
...     print x,
...
[0 3 1 4 2 5]
```

Iterating as a Specific Data Type

There are times when it is necessary to treat an array as a different data type than it is stored as. For instance, one may want to do all computations on 64-bit floats, even if the arrays being manipulated are 32-bit floats. Except when writing low-level C code, it's generally better to let the iterator handle the copying or buffering instead of casting the data type yourself in the inner loop.

There are two mechanisms which allow this to be done, temporary copies and buffering mode. With temporary copies, a copy of the entire array is made with the new data type, then iteration is done in the copy. Write access is permitted through a mode which updates the

original array after all the iteration is complete. The major drawback of temporary copies is that the temporary copy may consume a large amount of memory, particularly if the iteration data type has a larger itemsize than the original one.

Buffering mode mitigates the memory usage issue and is more cache-friendly than making temporary copies. Except for special cases, where the whole array is needed at once outside the iterator, buffering is recommended over temporary copying. Within NumPy, buffering is used by the ufuncs and other functions to support flexible inputs with minimal memory overhead.

In our examples, we will treat the input array with a complex data type, so that we can take square roots of negative numbers. Without enabling copies or buffering mode, the iterator will raise an exception if the data type doesn't match precisely.

Example:

```
>>> a = np.arange(6).reshape(2,3) - 3
>>> for x in np.nditer(a, op_dtypes=['complex128']):
...     print np.sqrt(x),
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Iterator operand required copying or buffering, but neither
copying nor buffering was enabled
```

In copying mode, 'copy' is specified as a per-operand flag. This is done to provide control in a per-operand fashion. Buffering mode is specified as an iterator flag.

Example:

```
>>> a = np.arange(6).reshape(2,3) - 3
>>> for x in np.nditer(a, op_flags=['readonly','copy'],
...                 op_dtypes=['complex128']):
...     print np.sqrt(x),
...
1.73205080757j 1.41421356237j 1j 0j (1+0j) (1.41421356237+0j)


>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['complex128']
):
...     print np.sqrt(x),
...
1.73205080757j 1.41421356237j 1j 0j (1+0j) (1.41421356237+0j)
```

The iterator uses NumPy's casting rules to determine whether a specific conversion is permitted. By default, it enforces 'safe' casting. This means, for example, that it will raise an exception if you try to treat a 64-bit float array as a 32-bit float array. In many cases, the rule 'same_kind' is the most reasonable rule to use, since it will allow conversion from 64 to 32-bit float, but not from float to int or from complex to float.

Example:

```
>>> a = np.arange(6.)
>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['float32']):
...     print x,
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Iterator operand 0 dtype could not be cast from dtype('floa
t64') to dtype('float32') according to the rule 'safe'


>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['float32'],
...                    casting='same_kind'):
...     print x,
...
0.0 1.0 2.0 3.0 4.0 5.0


>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['int32'], cas
ting='same_kind'):
...     print x,
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Iterator operand 0 dtype could not be cast from dtype('floa
t64') to dtype('int32') according to the rule 'same_kind'
```

One thing to watch out for is conversions back to the original data type when using a read-write or write-only operand. A common case is to implement the inner loop in terms of 64-bit floats, and use ‘same_kind’ casting to allow the other floating-point types to be processed as well. While in read-only mode, an integer array could be provided, read-write mode will raise an exception because conversion back to the array would violate the casting rule.

Example:

```
>>> a = np.arange(6)
>>> for x in np.nditer(a, flags=['buffered'], op_flags=['readwrite'],
...                    op_dtypes=['float64'], casting='same_kind'):
...     x[...] = x / 2.0
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: Iterator requested dtype could not be cast from dtype('floa
t64') to dtype('int64'), the operand 0 dtype, according to the rule 's
ame_kind'
```

Broadcasting Array Iteration

NumPy has a set of rules for dealing with arrays that have differing shapes which are applied whenever functions take multiple operands which combine element-wise. This is called broadcasting (ufuncs.html#ufuncs-broadcasting). The `nditer` (generated/numpy.nditer.html#numpy.nditer) object can apply these rules for you when you need to write such a function.

As an example, we print out the result of broadcasting a one and a two dimensional array together.

Example:

```
>>> a = np.arange(3)
>>> b = np.arange(6).reshape(2,3)
>>> for x, y in np.nditer([a,b]):
...     print "%d:%d" % (x,y),
...
0:0 1:1 2:2 0:3 1:4 2:5
```

When a broadcasting error occurs, the iterator raises an exception which includes the input shapes to help diagnose the problem.

Example:

```
>>> a = np.arange(2)
>>> b = np.arange(6).reshape(2,3)
>>> for x, y in np.nditer([a,b]):
...     print "%d:%d" % (x,y),
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (2) (
2,3)
```

Iterator-Allocated Output Arrays

A common case in NumPy functions is to have outputs allocated based on the broadcasting of the input, and additionally have an optional parameter called ‘out’ where the result will be placed when it is provided. The `nditer` (generated/numpy.nditer.html#numpy.nditer) object provides a convenient idiom that makes it very easy to support this mechanism.

We’ll show how this works by creating a function `square` (generated/numpy.square.html#numpy.square) which squares its input. Let’s start with a minimal function definition excluding ‘out’ parameter support.

Example:

```
>>> def square(a):
...     it = np.nditer([a, None])
...     for x, y in it:
...         y[...] = x*x
...     return it.operands[1]
...
>>> square([1,2,3])
array([1, 4, 9])
```

By default, the `nditer` (generated/numpy.nditer.html#numpy.nditer) uses the flags 'allocate' and 'writeonly' for operands that are passed in as None. This means we were able to provide just the two operands to the iterator, and it handled the rest.

When adding the 'out' parameter, we have to explicitly provide those flags, because if someone passes in an array as 'out', the iterator will default to 'readonly', and our inner loop would fail. The reason 'readonly' is the default for input arrays is to prevent confusion about unintentionally triggering a reduction operation. If the default were 'readwrite', any broadcasting operation would also trigger a reduction, a topic which is covered later in this document.

While we're at it, let's also introduce the 'no_broadcast' flag, which will prevent the output from being broadcast. This is important, because we only want one input value for each output. Aggregating more than one input value is a reduction operation which requires special handling. It would already raise an error because reductions must be explicitly enabled in an iterator flag, but the error message that results from disabling broadcasting is much more understandable for end-users. To see how to generalize the square function to a reduction, look at the sum of squares function in the section about Cython.

For completeness, we'll also add the 'external_loop' and 'buffered' flags, as these are what you will typically want for performance reasons.

Example:

```
>>> def square(a, out=None):
...     it = np.nditer([a, out],
...             flags = ['external_loop', 'buffered'],
...             op_flags = [['readonly'],
...                         ['writeonly', 'allocate', 'no_broadcast']]
)
...     for x, y in it:
...         y[...] = x*x
...     return it.operands[1]
...


>>> square([1,2,3])
array([1, 4, 9])
```

```
>>> b = np.zeros((3,))
>>> square([1,2,3], out=b)
array([ 1.,  4.,  9.])
>>> b
array([ 1.,  4.,  9.])


>>> square(np.arange(6).reshape(2,3), out=b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in square
ValueError: non-broadcastable output operand with shape (3) doesn't ma
tch the broadcast shape (2,3)
```

Outer Product Iteration

Any binary operation can be extended to an array operation in an outer product fashion like in `outer` (generated/numpy.outer.html#numpy.outer), and the `nditer` (generated/numpy.nditer.html#numpy.nditer) object provides a way to accomplish this by explicitly mapping the axes of the operands. It is also possible to do this with `newaxis` (arrays.indexing.html#numpy.newaxis) indexing, but we will show you how to directly use the nditer *op_axes* parameter to accomplish this with no intermediate views.

We'll do a simple outer product, placing the dimensions of the first operand before the dimensions of the second operand. The *op_axes* parameter needs one list of axes for each operand, and provides a mapping from the iterator's axes to the axes of the operand.

Suppose the first operand is one dimensional and the second operand is two dimensional. The iterator will have three dimensions, so *op_axes* will have two 3-element lists. The first list picks out the one axis of the first operand, and is -1 for the rest of the iterator axes, with a final result of [0, -1, -1]. The second list picks out the two axes of the second operand, but shouldn't overlap with the axes picked out in the first operand. Its list is [-1, 0, 1]. The output operand maps onto the iterator axes in the standard manner, so we can provide None instead of constructing another list.

The operation in the inner loop is a straightforward multiplication. Everything to do with the outer product is handled by the iterator setup.

Example:

```
>>> a = np.arange(3)
>>> b = np.arange(8).reshape(2,4)
>>> it = np.nditer([a, b, None], flags=['external_loop'],
...             op_axes=[[0, -1, -1], [-1, 0, 1], None])
>>> for x, y, z in it:
...     z[...] = x*y
...
>>> it.operands[2]
array([[[ 0,  0,  0,  0],
        [ 0,  0,  0,  0]],
       [[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 0,  2,  4,  6],
        [ 8, 10, 12, 14]]])
```

Reduction Iteration

Whenever a writeable operand has fewer elements than the full iteration space, that operand is undergoing a reduction. The `nditer` (generated/numpy.nditer.html#numpy.nditer) object requires that any reduction operand be flagged as read-write, and only allows reductions when ‘reduce_ok’ is provided as an iterator flag.

For a simple example, consider taking the sum of all elements in an array.

Example:

```
>>> a = np.arange(24).reshape(2,3,4)
>>> b = np.array(0)
>>> for x, y in np.nditer([a, b], flags=['reduce_ok', 'external_loop']
,
...                       op_flags=[['readonly'], ['readwrite']]):
...     y[...] += x
...
>>> b
array(276)
>>> np.sum(a)
276
```

Things are a little bit more tricky when combining reduction and allocated operands. Before iteration is started, any reduction operand must be initialized to its starting values. Here’s how we can do this, taking sums along the last axis of *a*.

Example:

```
>>> a = np.arange(24).reshape(2,3,4)
>>> it = np.nditer([a, None], flags=['reduce_ok', 'external_loop'],
...             op_flags=[['readonly'], ['readwrite', 'allocate']],
...             op_axes=[None, [0,1,-1]])
>>> it.operands[1][...] = 0
>>> for x, y in it:
...     y[...] += x
...
>>> it.operands[1]
array([[ 6, 22, 38],
       [54, 70, 86]])
>>> np.sum(a, axis=2)
array([[ 6, 22, 38],
       [54, 70, 86]])
```

To do buffered reduction requires yet another adjustment during the setup. Normally the iterator construction involves copying the first buffer of data from the readable arrays into the buffer. Any reduction operand is readable, so it may be read into a buffer. Unfortunately, initialization of the operand after this buffering operation is complete will not be reflected in the buffer that the iteration starts with, and garbage results will be produced.

The iterator flag “delay_bufalloc” is there to allow iterator-allocated reduction operands to exist together with buffering. When this flag is set, the iterator will leave its buffers uninitialized until it receives a reset, after which it will be ready for regular iteration. Here’s how the previous example looks if we also enable buffering.

Example:

```
>>> a = np.arange(24).reshape(2,3,4)
>>> it = np.nditer([a, None], flags=['reduce_ok', 'external_loop',
...                                  'buffered', 'delay_bufalloc'],
...             op_flags=[['readonly'], ['readwrite', 'allocate']],
...             op_axes=[None, [0,1,-1]])
>>> it.operands[1][...] = 0
>>> it.reset()
>>> for x, y in it:
...     y[...] += x
...
>>> it.operands[1]
array([[ 6, 22, 38],
       [54, 70, 86]])
```

Putting the Inner Loop in Cython

Those who want really good performance out of their low level operations should strongly consider directly using the iteration API provided in C, but for those who are not comfortable with C or C++, Cython is a good middle ground with reasonable performance tradeoffs. For the `nditer` (generated/numpy.nditer.html#numpy.nditer) object, this means letting the iterator take care of broadcasting, dtype conversion, and buffering, while giving the inner loop to Cython.

For our example, we'll create a sum of squares function. To start, let's implement this function in straightforward Python. We want to support an 'axis' parameter similar to the numpy `sum` (generated/numpy.sum.html#numpy.sum) function, so we will need to construct a list for the *op_axes* parameter. Here's how this looks.

Example:

```
>>> def axis_to_axeslist(axis, ndim):
...     if axis is None:
...         return [-1] * ndim
...     else:
...         if type(axis) is not tuple:
...             axis = (axis,)
...         axeslist = [1] * ndim
...         for i in axis:
...             axeslist[i] = -1
...         ax = 0
...         for i in range(ndim):
...             if axeslist[i] != -1:
...                 axeslist[i] = ax
...                 ax += 1
...         return axeslist
...
>>> def sum_squares_py(arr, axis=None, out=None):
...     axeslist = axis_to_axeslist(axis, arr.ndim)
...     it = np.nditer([arr, out], flags=['reduce_ok', 'external_loop'
,
...                                       'buffered', 'delay_bufalloc'
],
...                 op_flags=[['readonly'], ['readwrite', 'allocate']]
...                 op_axes=[None, axeslist],
...                 op_dtypes=['float64', 'float64'])
...     it.operands[1][...] = 0
...     it.reset()
...     for x, y in it:
...         y[...] += x*x
...     return it.operands[1]
...
>>> a = np.arange(6).reshape(2,3)
>>> sum_squares_py(a)
array(55.0)
>>> sum_squares_py(a, axis=-1)
array([  5.,  50.])
```

To Cython-ize this function, we replace the inner loop (y[...] += x*x) with Cython code that's specialized for the float64 dtype. With the 'external_loop' flag enabled, the arrays provided to the inner loop will always be one-dimensional, so very little checking needs to be done.

Here's the listing of sum_squares.pyx:

```
import numpy as np
cimport numpy as np
cimport cython

def axis_to_axeslist(axis, ndim):
    if axis is None:
        return [-1] * ndim
    else:
        if type(axis) is not tuple:
            axis = (axis,)
        axeslist = [1] * ndim
        for i in axis:
            axeslist[i] = -1
        ax = 0
        for i in range(ndim):
            if axeslist[i] != -1:
                axeslist[i] = ax
                ax += 1
        return axeslist

@cython.boundscheck(False)
def sum_squares_cy(arr, axis=None, out=None):
    cdef np.ndarray[double] x
    cdef np.ndarray[double] y
    cdef int size
    cdef double value

    axeslist = axis_to_axeslist(axis, arr.ndim)
    it = np.nditer([arr, out], flags=['reduce_ok', 'external_loop',
                                      'buffered', 'delay_bufalloc'],
               op_flags=[['readonly'], ['readwrite', 'allocate']],
               op_axes=[None, axeslist],
               op_dtypes=['float64', 'float64'])
    it.operands[1][...] = 0
    it.reset()
    for xarr, yarr in it:
        x = xarr
        y = yarr
        size = x.shape[0]
        for i in range(size):
           value = x[i]
           y[i] = y[i] + value * value
    return it.operands[1]
```

On this machine, building the .pyx file into a module looked like the following, but you may have to find some Cython tutorials to tell you the specifics for your system configuration.:

```
$ cython sum_squares.pyx
$ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -I/usr/include/python2.7
-fno-strict-aliasing -o sum_squares.so sum_squares.c
```

Running this from the Python interpreter produces the same answers as our native Python/NumPy code did.

Example:

```
>>> from sum_squares import sum_squares_cy
>>> a = np.arange(6).reshape(2,3)
>>> sum_squares_cy(a)
array(55.0)
>>> sum_squares_cy(a, axis=-1)
array([  5.,  50.])
```

Doing a little timing in IPython shows that the reduced overhead and memory allocation of the Cython inner loop is providing a very nice speedup over both the straightforward Python code and an expression using NumPy's built-in sum function.:

```
>>> a = np.random.rand(1000,1000)

>>> timeit sum_squares_py(a, axis=-1)
10 loops, best of 3: 37.1 ms per loop

>>> timeit np.sum(a*a, axis=-1)
10 loops, best of 3: 20.9 ms per loop

>>> timeit sum_squares_cy(a, axis=-1)
100 loops, best of 3: 11.8 ms per loop

>>> np.all(sum_squares_cy(a, axis=-1) == np.sum(a*a, axis=-1))
True

>>> np.all(sum_squares_py(a, axis=-1) == np.sum(a*a, axis=-1))
True
```

Table Of Contents (../contents.html)

Previous topic

Indexing (arrays.indexing.html)

Next topic

Standard array subclasses (arrays.classes.html)