Iterating Over Arrays

在NumPy 1.6中引入的迭代器对象`nditer` (generated/numpy.nditer.html#numpy.nditer)提供了以系统方式访问一个或多个数组的所有元素的许多灵活方式。本页介绍了使用对象在Python中的数组计算的一些基本方法，然后得出结论，如何加速Cython中的内循环。由于`nditer` (generated/numpy.nditer.html#numpy.nditer)的Python暴露是C数组迭代器API的相对直接的映射，这些想法还将提供帮助处理C或C ++的数组迭代。

Single Array Iteration

使用`nditer` (generated/numpy.nditer.html#numpy.nditer)可以完成的最基本的任务是访问数组的每个元素。使用标准的Python迭代器接口逐个提供每个元素。

例:

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

对于这个迭代，要注意的一个重要的事情是，选择顺序以匹配数组的存储器布局，而不是使用标准C或Fortran排序。这是为了访问效率，反映这样的想法，默认情况下，只是想访问每个元素，而不关心特定的顺序。我们可以通过迭代我们以前的数组的转置来看到这一点，与以C顺序取转置的副本相比。

例:

```
>>> 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
```

*a*和*aT*的元素以相同的顺序遍历，即它们存储在内存中的顺序，而*元素aTcopy (order = C')* 以不同的顺序访问，因为它们已被放入不同的内存布局。

Controlling Iteration Order

有时，以特定顺序访问数组的元素很重要，而不考虑内存中元素的布局。`nditer` (generated/numpy.nditer.html#numpy.nditer)对象提供*顺序*参数以控制迭代的此方面。具有上述行为的默认值是order ='K'以保持现有顺序。对于C order，可以用order ='C'；对于Fortran order，可以覆盖order ='F'。

例:

```
>>> 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

默认情况下，`nditer` (generated/numpy.nditer.html#numpy.nditer)将输入数组视为只读对象。要修改数组元素，您必须指定读写或只写模式。这由每操作数标志控制。

在Python中的常规赋值只是改变局部或全局变量字典中的引用，而不是原地修改现有变量。这意味着，简单地赋值给*x*不会将该值放入数组的元素，而是将*x*从数组元素引用切换为引用该值你分配。要实际修改数组的元素，应将*x*与省略号建立索引。

例:

```
>>> 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

在到目前为止的所有示例中，*a*的元素由迭代器一次提供一个，因为所有循环逻辑都在迭代器内部。虽然这是简单和方便，它不是很有效率。一个更好的方法是将一维最内层循环移到你的代码中，在迭代器外部。这样，NumPy的矢量化操作可以用在被访问的元素的更大的块上。

`nditer` (generated/numpy.nditer.html#numpy.nditer)将尝试向内部循环提供尽可能大的块。通过强制'C'和'F'顺序，我们得到不同的外部循环大小。通过指定迭代器标志来启用此模式。

观察到默认情况下保持原生内存顺序，迭代器能够提供单个一维块，而当强制Fortran顺序时，它必须提供三个块的两个元素。

例:

```
>>> 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

在迭代期间，您可能想在计算中使用当前元素的索引。例如，您可能希望按内存顺序访问数组的元素，但使用C顺序，Fortran顺序或多维索引来查找不同数组中的值。

Python迭代器协议没有从迭代器查询这些附加值的自然方式，因此我们引入了一个用于使用`nditer` (generated/numpy.nditer.html#numpy.nditer)进行迭代的替代语法。此语法显式地与迭代器对象本身一起使用，因此其属性在迭代期间可以轻松访问。使用这个循环结构，可以通过索引到迭代器来访问当前值，并且正在跟踪的索引是属性*index*或*multi_index*，具体取决于请求的内容。

Python交互式解释器不幸地在循环的每次迭代期间在while循环中打印出表达式的值。我们使用这个循环结构修改了示例中的输出，以便更加可读。

例:

```
>>> 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]])
```

跟踪索引或多索引与使用外部循环不兼容，因为它需要每个元素有不同的索引值。如果你尝试合并这些标志，`nditer` (generated/numpy.nditer.html#numpy.nditer)对象将引发异常

例:

```
>>> 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

当强制迭代顺序时，我们观察到外部循环选项可以提供较小块中的元素，因为不能以恒定的步长以适当的顺序访问元素。当编写C代码时，这通常很好，但是在纯Python代码中，这可能导致性能的显着降低。

通过启用缓冲模式，迭代器提供给内部循环的块可以做得更大，从而显着降低Python解释器的开销。在强制Fortran迭代顺序的示例中，内部循环在启用缓冲时可以一次查看所有元素。

例:

```
>>> 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

有时，有必要将数组视为与存储为不同的数据类型。例如，可能想对64位浮点数执行所有计算，即使正在操作的数组是32位浮点数。除了编写低级C代码之外，通常最好让迭代器处理复制或缓冲，而不是自己在内循环中转换数据类型。

有两种机制允许这样做，临时副本和缓冲模式。使用临时副本，使用新数据类型创建整个数组的副本，然后在副本中进行迭代。允许通过在所有迭代完成后更新原始数组的模式进行写访问。临时副本的主要缺点是临时副本可能消耗大量的内存，特别是如果迭代数据类型具有比原始数据类型更大的项目大小。

缓冲模式减少了内存使用问题，并且比临时副本更容易缓存。除了特殊情况，其中整个数组一次在迭代器外部需要，建议在临时复制时进行缓冲。在NumPy中，缓冲由ufuncs和其他函数使用，以最小的内存开销支持灵活的输入。

在我们的例子中，我们将使用复杂数据类型来处理输入数组，以便我们可以取负数的平方根。如果不启用复制或缓冲模式，如果数据类型不精确匹配，迭代器将引发异常。

例:

```
>>> 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 e
nabled
```

在复制模式下，'copy'被指定为每操作数标志。这是为了以操作数方式提供控制。缓冲模式被指定为迭代器标志。

例:

```
>>> 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)
```

迭代器使用NumPy的投射规则来确定是否允许特定的转化。默认情况下，它强制执行“安全”转换。这意味着，例如，如果您尝试将64位浮点数组视为32位浮点数组，它将引发异常。在许多情况下，规则“same_kind”是使用的最合理的规则，因为它将允许从64位转换为32位浮点，但不能从float转换为int或从complex转换为float。

例:

```
>>> 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('float64') to dtype('float32') accor
ding 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'], casting='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('float64') to dtype('int32') accordi
ng to the rule 'same_kind'
```

需要注意的一点是，当使用读写或只写操作数时，转换回原始数据类型。一种常见的情况是以64位浮点数实现内部循环，并使用“same_kind”转换来允许处理其他浮点类型。在只读模式下，可以提供一个整数数组，读写模式将引发异常，因为转换回数组将违反转换规则。

例:

```
>>> 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('float64') to dtype('int64'), the ope
rand 0 dtype, according to the rule 'same_kind'
```

Broadcasting Array Iteration

NumPy有一组用于处理数组的规则，它们具有不同的形状，每当函数采用多个以元素方式组合的操作数时。这称为broadcasting (ufuncs.html#ufuncs-broadcasting)。当你需要写这样的函数时，`nditer` (generated/numpy.nditer.html#numpy.nditer)对象可以应用这些规则给你。

作为示例，我们打印出一个一维和二维数组一起广播的结果。

例:

```
>>> 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
```

当发生广播错误时，迭代器引发包括输入形状的异常，以帮助诊断问题。

例:

```
>>> 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

NumPy函数中的一个常见情况是具有基于输入的广播分配的输出，并且另外具有称为“out”的可选参数，其中当提供结果时将其放置。 `nditer` (generated/numpy.nditer.html#numpy.nditer)对象提供了一个方便的习语，使其非常容易支持此机制。

我们将通过创建一个平方其输入的函数 `square` (generated/numpy.square.html#numpy.square)来显示这是如何工作的。让我们从一个最小的函数定义开始，不包括'out'参数支持。

例:

```
>>> 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])
```

默认情况下， `nditer` (generated/numpy.nditer.html#numpy.nditer)对于作为None传入的操作数使用标志“allocate”和“writeonly”。这意味着我们能够为迭代器提供两个操作数，并处理其余的操作数。

当添加'out'参数时，我们必须显式提供这些标志，因为如果有人传入一个数组作为'out'，迭代器将默认为'readonly'，我们的内循环将失败。“readonly”的原因是输入数组的默认值是为了防止关于无意触发归约操作的混淆。如果默认值为“readwrite”，则任何广播操作都将触发缩减，本文档后面将介绍一个主题。

虽然我们在这里，我们还介绍了“no_broadcast”标志，这将防止输出的广播。这很重要，因为每个输出只需要一个输入值。聚合多个输入值是需要特殊处理的缩减操作。它已经引起一个错误，因为必须在迭代器标志中显式地启用缩减，但是禁用广播导致的错误消息对于最终用户来说更容易理解。要了解如何将平方函数泛化为简化，请参阅有关Cython的部分中的平方和函数。

为了完整性，我们还将添加“external_loop”和“buffered”标志，因为这些是你通常因为性能原因而需要的。

例:

```
>>> 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 match the broadcast shape
 (2,3)
```

Outer Product Iteration

任何二进制操作都可以以外部乘积方式扩展到数组操作，如`outer`(generated/numpy.outer.html#numpy.outer)，而`nditer`(generated/numpy.nditer.html#numpy.nditer)对象提供了一种实现方法，操作数。也可以使用`newaxis`(arrays.indexing.html#numpy.newaxis)索引来实现这一点，但是我们将告诉你如何直接使用nditer *op_axes*参数来实现这一点，没有中间视图。

我们将做一个简单的外部产品，将第一个操作数的维度放在第二个操作数的维度之前。*op_axes*参数需要每个操作数的一个轴列表，并提供从迭代器轴到操作数轴的映射。

假设第一操作数是一维的，而第二操作数是二维的。迭代器将有三个维度，因此*op_axes*将有两个3元素列表。第一个列表选取第一个操作数的一个轴，对于其余的迭代器轴为-1，最终结果为[0，-1，-1]。第二个列表拾取第二个操作数的两个轴，但不应与第一个操作数中拾取的轴重叠。它的列表是[-1,0,1]。输出操作数以标准方式映射到迭代器轴，因此我们可以提供None而不是构造另一个列表。

内循环中的操作是直接乘法。与外部产品相关的一切都由迭代器设置来处理。

例:

```
>>> 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

每当可写操作数具有比完全迭代空间少的元素时，该操作数正在经历减少。`nditer`(generated/numpy.nditer.html#numpy.nditer)对象要求将任何还原操作数标记为读写，并且仅当“reduce_ok”作为迭代器标志提供时才允许减少。

对于一个简单的例子，考虑数组中所有元素的和。

例:

```
>>> 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
```

当组合缩减和分配的操作数时，事情有点棘手。在开始迭代之前，任何归约操作数必须初始化为其起始值。我们可以这样做，沿着*a*的最后一个轴取和。

例:

```
>>> 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]])
```

要进行缓冲减少，需要在设置期间进行另一个调整。通常，迭代器构造涉及将数据的第一缓冲器从可读数组复制到缓冲器中。任何缩减操作数是可读的，因此它可以被读入缓冲器。不幸的是，在这个缓冲操作完成之后，操作数的初始化将不会反映在迭代开始的缓冲器中，并且将产生垃圾结果。

当设置此标志时，迭代器将保持其缓冲区未初始化，直到它接收到复位，之后它将准备好进行常规迭代。如果我们也启用缓冲，上面的例子看起来如下。

例:

```
>>> 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

那些想要在低级操作中表现出色的用户应该强烈地考虑直接使用C中提供的迭代API，但是对于那些对C或C ++不熟悉的用户来说，Cython是一个很好的中间点，并且有着合理的性能权衡。对于 `nditer` (generated/numpy.nditer.html#numpy.nditer)对象，这意味着让迭代器处理广播，dtype转换和缓冲，同时给Cython提供内循环。

在我们的例子中，我们将创建一个平方和函数。首先，让我们使用简单的Python实现这个函数。我们要支持类似于numpy `sum` (generated/numpy.sum.html#numpy.sum)函数的“axis”参数，因此我们需要为*op_axes*参数构造一个列表。这是这个样子。

例:

```
>>> 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.])
```

为了Cython -ize这个函数，我们用专用于float64 dtype的Cython代码替换内部循环（y [...] + = x * x）。如果启用了'external_loop'标志，提供给内循环的数组将始终是一维的，因此需要进行非常少的检查。

以下是sum_squares.pyx的列表：

```
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]
```

在这台机器上，将.pyx文件构建为一个模块看起来像下面这样，但是你可能需要找到一些Cython教程来告诉你系统配置的细节。 ：

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

从Python解释器运行它产生与我们的本地Python / NumPy代码相同的答案。

例:

```
>>> 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.])
```

在IPython中做一点时间表明，减少的Cython内部循环的开销和内存分配提供了一个非常好的加速，超过直接的Python代码和使用NumPy的内置sum函数的表达式。：

```
>>> 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
```