Indexing

See also:
Indexing routines (../reference/routines.indexing.html#routines-indexing)

Array indexing refers to any use of the square brackets ([]) to index array values. There are many options to indexing, which give numpy indexing great power, but with power comes some complexity and the potential for confusion. This section is just an overview of the various options and issues related to indexing. Aside from single element indexing, the details on most of these options are to be found in related sections.

Assignment vs referencing

Most of the following examples show the use of indexing when referencing data in an array. The examples work just as well when assigning to an array. See the section at the end for specific examples and explanations on how assignments work.

Single element indexing

Single element indexing for a 1-D array is what one expects. It work exactly like that for other standard Python sequences. It is 0-based, and accepts negative indices for indexing from the end of the array.

```
>>> x = np.arange(10)
>>> x[2]
2
>>> x[-2]
8
```

Unlike lists and tuples, numpy arrays support multidimensional indexing for multidimensional arrays. That means that it is not necessary to separate each dimension's index into its own set of square brackets.

```
>>> x.shape = (2,5) # now x is 2-dimensional
>>> x[1,3]
8
>>> x[1,-1]
9
```

Note that if one indexes a multidimensional array with fewer indices than dimensions, one gets a subdimensional array. For example:

```
>>> x[0]
array([0, 1, 2, 3, 4])
```

That is, each index specified selects the array corresponding to the rest of the dimensions selected. In the above example, choosing 0 means that the remaining dimension of length 5 is being left unspecified, and that what is returned is an array of that dimensionality and size. It must be noted that the returned array is not a copy of the original, but points to the same values in memory as does the original array. In this case, the 1-D array at the first position (0) is returned. So using a single index on the returned array, results in a single element being returned. That is:

```
>>> x[0][2]
2
```

So note that `x[0,2] = x[0][2]` though the second case is more inefficient as a new temporary array is created after the first index that is subsequently indexed by 2.

Note to those used to IDL or Fortran memory order as it relates to indexing. NumPy uses C-order indexing. That means that the last index usually represents the most rapidly changing memory location, unlike Fortran or IDL, where the first index represents the most rapidly changing location in memory. This difference represents a great potential for confusion.

Other indexing options

It is possible to slice and stride arrays to extract arrays of the same number of dimensions, but of different sizes than the original. The slicing and striding works exactly the same way it does for lists and tuples except that they can be applied to multiple dimensions as well. A few examples illustrates best:

```
>>> x = np.arange(10)
>>> x[2:5]
array([2, 3, 4])
>>> x[:-7]
array([0, 1, 2])
>>> x[1:7:2]
array([1, 3, 5])
>>> y = np.arange(35).reshape(5,7)
>>> y[1:5:2,::3]
array([[ 7, 10, 13],
       [21, 24, 27]])
```

Note that slices of arrays do not copy the internal array data but also produce new views of the original data.

It is possible to index arrays with other arrays for the purposes of selecting lists of values out of arrays into new arrays. There are two different ways of accomplishing this. One uses one or more arrays of index values. The other involves giving a boolean array of the proper shape to indicate the values to be selected. Index arrays are a very powerful tool that allow one to avoid looping over individual elements in arrays and thus greatly improve performance.

It is possible to use special features to effectively increase the number of dimensions in an array through indexing so the resulting array aquires the shape needed for use in an expression or with a specific function.

Index arrays

NumPy arrays may be indexed with other arrays (or any other sequence- like object that can be converted to an array, such as lists, with the exception of tuples; see the end of this document for why this is). The use of index arrays ranges from simple, straightforward cases to complex, hard-to-understand cases. For all cases of index arrays, what is returned is a copy of the original data, not a view as one gets for slices.

Index arrays must be of integer type. Each value in the array indicates which value in the array to use in place of the index. To illustrate:

```
>>> x = np.arange(10,1,-1)
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[np.array([3, 3, 1, 8])]
array([7, 7, 9, 2])
```

The index array consisting of the values 3, 3, 1 and 8 correspondingly create an array of length 4 (same as the index array) where each index is replaced by the value the index array has in the array being indexed.

Negative values are permitted and work as they do with single indices or slices:

```
>>> x[np.array([3,3,-3,8])]
array([7, 7, 4, 2])
```

It is an error to have index values out of bounds:

```
>>> x[np.array([3, 3, 20, 8])]
<type 'exceptions.IndexError'>: index 20 out of bounds 0<=index<9
```

Generally speaking, what is returned when index arrays are used is an array with the same shape as the index array, but with the type and values of the array being indexed. As an example, we can use a multidimensional index array instead:

```
>>> x[np.array([[1,1],[2,3]])]
array([[9, 9],
       [8, 7]])
```

Indexing Multi-dimensional arrays

Things become more complex when multidimensional arrays are indexed, particularly with multidimensional index arrays. These tend to be more unusual uses, but they are permitted, and they are useful for some problems. We'll start with the simplest multidimensional case (using the array y from the previous examples):

```
>>> y[np.array([0,2,4]), np.array([0,1,2])]
array([ 0, 15, 30])
```

In this case, if the index arrays have a matching shape, and there is an index array for each dimension of the array being indexed, the resultant array has the same shape as the index arrays, and the values correspond to the index set for each position in the index arrays. In this example, the first index value is 0 for both index arrays, and thus the first value of the resultant array is y[0,0]. The next value is y[2,1], and the last is y[4,2].

If the index arrays do not have the same shape, there is an attempt to broadcast them to the same shape. If they cannot be broadcast to the same shape, an exception is raised:

```
>>> y[np.array([0,2,4]), np.array([0,1])]
<type 'exceptions.ValueError'>: shape mismatch: objects cannot be
broadcast to a single shape
```

The broadcasting mechanism permits index arrays to be combined with scalars for other indices. The effect is that the scalar value is used for all the corresponding values of the index arrays:

```
>>> y[np.array([0,2,4]), 1]
array([ 1, 15, 29])
```

Jumping to the next level of complexity, it is possible to only partially index an array with index arrays. It takes a bit of thought to understand what happens in such cases. For example if we just use one index array with y:

```
>>> y[np.array([0,2,4])]
array([[ 0,  1,  2,  3,  4,  5,  6],
       [14, 15, 16, 17, 18, 19, 20],
       [28, 29, 30, 31, 32, 33, 34]])
```

What results is the construction of a new array where each value of the index array selects one row from the array being indexed and the resultant array has the resulting shape (number of index elements, size of row).

An example of where this may be useful is for a color lookup table where we want to map the values of an image into RGB triples for display. The lookup table could have a shape (nlookup, 3). Indexing such an array with an image with shape (ny, nx) with dtype=np.uint8 (or any integer type so long as values are with the bounds of the lookup table) will result in an array of shape (ny, nx, 3) where a triple of RGB values is associated with each pixel location.

In general, the shape of the resultant array will be the concatenation of the shape of the index array (or the shape that all the index arrays were broadcast to) with the shape of any unused dimensions (those not indexed) in the array being indexed.

Boolean or "mask" index arrays

Boolean arrays used as indices are treated in a different manner entirely than index arrays. Boolean arrays must be of the same shape as the initial dimensions of the array being indexed. In the most straightforward case, the boolean array has the same shape:

```
>>> b = y>20
>>> y[b]
array([21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34])
```

Unlike in the case of integer index arrays, in the boolean case, the result is a 1-D array containing all the elements in the indexed array corresponding to all the true elements in the boolean array. The elements in the indexed array are always iterated and returned in row-major (../glossary.html#term-row-major) (C-style) order. The result is also identical to `y[np.nonzero(b)]`. As with index arrays, what is returned is a copy of the data, not a view as one gets with slices.

The result will be multidimensional if y has more dimensions than b. For example:

```
>>> b[:,5] # use a 1-D boolean whose first dim agrees with the first dim of y
array([False, False, False,  True,  True])
>>> y[b[:,5]]
array([[21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])
```

Here the 4th and 5th rows are selected from the indexed array and combined to make a 2-D array.

In general, when the boolean array has fewer dimensions than the array being indexed, this is equivalent to y[b, ...], which means y is indexed by b followed by as many : as are needed to fill out the rank of y. Thus the shape of the result is one dimension containing the number of True elements of the boolean array, followed by the remaining dimensions of the array being indexed.

For example, using a 2-D boolean array of shape (2,3) with four True elements to select rows from a 3-D array of shape (2,3,5) results in a 2-D result of shape (4,5):

```
>>> x = np.arange(30).reshape(2,3,5)
>>> x
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14]],
       [[15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]])
>>> b = np.array([[True, True, False], [False, True, True]])
>>> x[b]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29]])
```

For further details, consult the numpy reference documentation on array indexing.

Combining index arrays with slices

Index arrays may be combined with slices. For example:

```
>>> y[np.array([0,2,4]),1:3]
array([[ 1,  2],
       [15, 16],
       [29, 30]])
```

In effect, the slice is converted to an index array np.array([[1,2]]) (shape (1,2)) that is broadcast with the index array to produce a resultant array of shape (3,2).

Likewise, slicing can be combined with broadcasted boolean indices:

```
>>> y[b[:,5],1:3]
array([[22, 23],
       [29, 30]])
```

Structural indexing tools

To facilitate easy matching of array shapes with expressions and in assignments, the np.newaxis object can be used within array indices to add new dimensions with a size of 1. For example:

```
>>> y.shape
(5, 7)
>>> y[:,np.newaxis,:].shape
(5, 1, 7)
```

Note that there are no new elements in the array, just that the dimensionality is increased. This can be handy to combine two arrays in a way that otherwise would require explicitly reshaping operations. For example:

```
>>> x = np.arange(5)
>>> x[:,np.newaxis] + x[np.newaxis,:]
array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7],
       [4, 5, 6, 7, 8]])
```

The ellipsis syntax maybe used to indicate selecting in full any remaining unspecified dimensions. For example:

```
>>> z = np.arange(81).reshape(3,3,3,3)
>>> z[1,...,2]
array([[29, 32, 35],
       [38, 41, 44],
       [47, 50, 53]])
```

This is equivalent to:

```
>>> z[1,:,:,2]
array([[29, 32, 35],
       [38, 41, 44],
       [47, 50, 53]])
```

Assigning values to indexed arrays

As mentioned, one can select a subset of an array to assign to using a single index, slices, and index and mask arrays. The value being assigned to the indexed array must be shape consistent (the same shape or broadcastable to the shape the index produces). For example, it is permitted to assign a constant to a slice:

```
>>> x = np.arange(10)
>>> x[2:7] = 1
```

or an array of the right size:

```
>>> x[2:7] = np.arange(5)
```

Note that assignments may result in changes if assigning higher types to lower types (like floats to ints) or even exceptions (assigning complex to floats or ints):

```
>>> x[1] = 1.2
>>> x[1]
1
>>> x[1] = 1.2j
<type 'exceptions.TypeError'>: can't convert complex to long; use
long(abs(z))
```

Unlike some of the references (such as array and mask indices) assignments are always made to the original data in the array (indeed, nothing else would make sense!). Note though, that some actions may not work as one may naively expect. This particular example is often surprising to people:

```
>>> x = np.arange(0, 50, 10)
>>> x
array([ 0, 10, 20, 30, 40])
>>> x[np.array([1, 1, 3, 1])] += 1
>>> x
array([ 0, 11, 20, 31, 40])
```

Where people expect that the 1st location will be incremented by 3. In fact, it will only be incremented by 1. The reason is because a new array is extracted from the original (as a temporary) containing the values at 1, 1, 3, 1, then the value 1 is added to the temporary, and then the temporary is assigned back to the original array. Thus the value of the array at x[1]+1 is assigned to x[1] three times, rather than being incremented 3 times.

Dealing with variable numbers of indices within programs

The index syntax is very powerful but limiting when dealing with a variable number of indices. For example, if you want to write a function that can handle arguments with various numbers of dimensions without having to write special case code for each number of possible dimensions, how can that be done? If one supplies to the index a tuple, the tuple will be interpreted as a list of indices. For example (using the previous definition for the array z):

```
>>> indices = (1,1,1,1)
>>> z[indices]
40
```

So one can use code to construct tuples of any number of indices and then use these within an index.

Slices can be specified within programs by using the slice() function in Python. For example:

```
>>> indices = (1,1,1,slice(0,2)) # same as [1,1,1,0:2]
>>> z[indices]
array([39, 40])
```

Likewise, ellipsis can be specified by code by using the Ellipsis object:

```
>>> indices = (1, Ellipsis, 1) # same as [1,...,1]
>>> z[indices]
array([[28, 31, 34],
       [37, 40, 43],
       [46, 49, 52]])
```

For this reason it is possible to use the output from the np.nonzero() function directly as an index since it always returns a tuple of index arrays.

Because the special treatment of tuples, they are not automatically converted to an array as a list would be. As an example:

```
>>> z[[1,1,1,1]] # produces a large array
array([[[[27, 28, 29],
         [30, 31, 32], ...
>>> z[(1,1,1,1)] # returns a single value
40
```