Indexing on `ndarrays`

See also

Indexing routines

`ndarrays` can be indexed using the standard Python `x[obj]` syntax, where *x* is the array and *obj* the selection. There are different kinds of indexing available depending on *obj*: basic indexing, advanced indexing and field access.

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 Assigning values to indexed arrays for specific examples and explanations on how assignments work.

Note that in Python, `x[(exp1, exp2, ..., expN)]` is equivalent to `x[exp1, exp2, ..., expN]`; the latter is just syntactic sugar for the former.

Basic indexing

Single element indexing

Single element indexing works 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
```

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 a view, i.e., it 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

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.

Slicing and striding

Basic slicing extends Python's basic concept of slicing to N dimensions. Basic slicing occurs when *obj* is a `slice` object (constructed by `start:stop:step` notation inside of brackets), an

integer, or a tuple of slice objects and integers. `Ellipsis` and `newaxis` objects can be interspersed with these as well.

The simplest case of indexing with *N* integers returns an array scalar representing the corresponding item. As in Python, all indices are zero-based: for the *i*-th index n_i, the valid range is $0 \le n_i < d_i$ where d_i is the *i*-th element of the shape of the array. Negative indices are interpreted as counting from the end of the array (*i.e.*, if $n_i < 0$, it means $n_i + d_i$).

All arrays generated by basic slicing are always views of the original array.

> **Note**
>
> NumPy slicing creates a view instead of a copy as in the case of built-in Python sequences such as string, tuple and list. Care must be taken when extracting a small portion from a large array which becomes useless after the extraction, because the small portion extracted contains a reference to the large original array whose memory will not be released until all arrays derived from it are garbage-collected. In such cases an explicit `copy()` is recommended.

The standard rules of sequence slicing apply to basic slicing on a per-dimension basis (including using a step index). Some useful concepts to remember include:

- The basic slice syntax is `i:j:k` where *i* is the starting index, *j* is the stopping index, and *k* is the step ($k\neq0$). This selects the *m* elements (in the corresponding dimension) with index values *i*, *i* + *k*, ..., *i* + *(m - 1) k* where $m = q + (r\neq0)$ and *q* and *r* are the quotient and remainder obtained by dividing *j* - *i* by *k*: *j* - *i* = *q k* + *r*, so that *i* + *(m - 1) k* < *j*. For example:

  ```
  >>> x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  >>> x[1:7:2]
  array([1, 3, 5])
  ```

- Negative *i* and *j* are interpreted as *n* + *i* and *n* + *j* where *n* is the number of elements in the corresponding dimension. Negative *k* makes stepping go towards smaller indices. From the above example:

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

- Assume *n* is the number of elements in the dimension being sliced. Then, if *i* is not given it defaults to 0 for *k > 0* and *n - 1* for *k < 0* . If *j* is not given it defaults to *n* for *k > 0* and *-n-1* for *k < 0* . If *k* is not given it defaults to 1. Note that `::` is the same as `:` and means select all indices along this axis. From the above example:

```
>>> x[5:]
array([5, 6, 7, 8, 9])
```

- If the number of objects in the selection tuple is less than *N*, then `:` is assumed for any subsequent dimensions. For example:

```
>>> x = np.array([[[1],[2],[3]], [[4],[5],[6]]])
>>> x.shape
(2, 3, 1)
>>> x[1:2]
array([[[4],
        [5],
        [6]]])
```

- An integer, *i*, returns the same values as `i:i+1` **except** the dimensionality of the returned object is reduced by 1. In particular, a selection tuple with the *p*-th element an integer (and all other entries `:`) returns the corresponding sub-array with dimension *N - 1*. If *N = 1* then the returned object is an array scalar. These objects are explained in Scalars.
- If the selection tuple has all entries `:` except the *p*-th entry which is a slice object `i:j:k`, then the returned array has dimension *N* formed by stacking, along the *p*-th axis, the sub-arrays returned by integer indexing of elements *i*, *i+k*, ..., *i + (m - 1) k < j*.
- Basic slicing with more than one non-`:` entry in the slicing tuple, acts like repeated application of slicing using a single non-`:` entry, where the non-`:` entries are successively taken (with all other non-`:` entries replaced by `:`). Thus, `x[ind1, ..., ind2,:]` acts like `x[ind1][..., ind2, :]` under basic slicing.

> **Warning**
>
> The above is **not** true for advanced indexing.

- You may use slicing to set values in the array, but (unlike lists) you can never grow the array. The size of the value to be set in `x[obj] = value` must be (broadcastable to) the same shape as `x[obj]`.
- A slicing tuple can always be constructed as *obj* and used in the `x[obj]` notation. Slice objects can be used in the construction in place of the `[start:stop:step]` notation. For example, `x[1:10:5, ::-1]` can also be implemented as `obj = (slice(1, 10, 5), slice(None, None, -1)); x[obj]` . This can be useful for constructing generic code that works on arrays of arbitrary dimensions. See Dealing with variable numbers of indices within programs for more information.

Dimensional indexing tools

There are some tools to facilitate the easy matching of array shapes with expressions and in assignments.

`Ellipsis` expands to the number of `:` objects needed for the selection tuple to index all dimensions. In most cases, this means that the length of the expanded selection tuple is `x.ndim`. There may only be a single ellipsis present. From the above example:

```
>>> x[..., 0]
array([[1, 2, 3],
       [4, 5, 6]])
```

This is equivalent to:

```
>>> x[:, :, 0]
array([[1, 2, 3],
       [4, 5, 6]])
```

Each `newaxis` object in the selection tuple serves to expand the dimensions of the resulting selection by one unit-length dimension. The added dimension is the position of the `newaxis` object in the selection tuple. `newaxis` is an alias for `None`, and `None` can be used in place of this with the same result. From the above example:

```
>>> x[:, np.newaxis, :, :].shape
(2, 1, 3, 1)
>>> x[:, None, :, :].shape
(2, 1, 3, 1)
```

This can be handy to combine two arrays in a way that otherwise would require explicit 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]])
```

Advanced indexing

Advanced indexing is triggered when the selection object, *obj*, is a non-tuple sequence object, an `ndarray` (of data type integer or bool), or a tuple with at least one sequence object or ndarray (of data type integer or bool). There are two types of advanced indexing: integer and Boolean.

Advanced indexing always returns a *copy* of the data (contrast with basic slicing that returns a view).

Warning

The definition of advanced indexing means that `x[(1, 2, 3),]` is fundamentally different than `x[(1, 2, 3)]`. The latter is equivalent to `x[1, 2, 3]` which will trigger basic selection while the former will trigger advanced indexing. Be sure to understand why this occurs.

Integer array indexing

Integer array indexing allows selection of arbitrary items in the array based on their *N*-dimensional index. Each integer array represents a number of indices into that dimension.

Negative values are permitted in the index arrays and work as they do with single indices or slices:

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

If the index values are out of bounds then an `IndexError` is thrown:

```
>>> x = np.array([[1, 2], [3, 4], [5, 6]])
>>> x[np.array([1, -1])]
array([[3, 4],
       [5, 6]])
>>> x[np.array([3, 4])]
Traceback (most recent call last):
  ...
IndexError: index 3 is out of bounds for axis 0 with size 3
```

When the index consists of as many integer arrays as dimensions of the array being indexed, the indexing is straightforward, but different from slicing.

Advanced indices always are broadcast and iterated as *one*:

```
result[i_1, ..., i_M] == x[ind_1[i_1, ..., i_M], ind_2[i_1, ..., i_M],
                           ..., ind_N[i_1, ..., i_M]]
```

Note that the resulting shape is identical to the (broadcast) indexing array shapes `ind_1, ..., ind_N`. If the indices cannot be broadcast to the same shape, an exception `IndexError: shape mismatch: indexing arrays could not be broadcast together with shapes...` is raised.

Indexing with multidimensional index arrays 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:

```
>>> y = np.arange(35).reshape(5, 7)
>>> y
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, 30, 31, 32, 33, 34]])
>>> 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])]
Traceback (most recent call last):
  ...
IndexError: shape mismatch: indexing arrays could not be broadcast
together with shapes (3,) (2,)
```

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]])
```

It results in 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).

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.

Example

From each row, a specific element should be selected. The row index is just `[0, 1, 2]` and the column index specifies the element to choose for the corresponding row, here `[0, 1, 0]`. Using both together the task can be solved using advanced indexing:

```
>>> x = np.array([[1, 2], [3, 4], [5, 6]])
>>> x[[0, 1, 2], [0, 1, 0]]
array([1, 4, 5])
```

To achieve a behaviour similar to the basic slicing above, broadcasting can be used. The function `ix_` can help with this broadcasting. This is best understood with an example.

Example

From a 4x3 array the corner elements should be selected using advanced indexing. Thus all elements for which the column is one of `[0, 2]` and the row is one of `[0, 3]` need to be selected. To use advanced indexing one needs to select all elements *explicitly*. Using the method explained previously one could write:

```
>>> x = np.array([[ 0,  1,  2],
...               [ 3,  4,  5],
...               [ 6,  7,  8],
...               [ 9, 10, 11]])
>>> rows = np.array([[0, 0],
...                  [3, 3]], dtype=np.intp)
>>> columns = np.array([[0, 2],
...                     [0, 2]], dtype=np.intp)
>>> x[rows, columns]
array([[ 0,  2],
       [ 9, 11]])
```

However, since the indexing arrays above just repeat themselves, broadcasting can be used (compare operations such as `rows[:, np.newaxis] + columns`) to simplify this:

```
>>> rows = np.array([0, 3], dtype=np.intp)
>>> columns = np.array([0, 2], dtype=np.intp)
>>> rows[:, np.newaxis]
array([[0],
       [3]])
>>> x[rows[:, np.newaxis], columns]
array([[ 0,  2],
       [ 9, 11]])
```

This broadcasting can also be achieved using the function `ix_`:

```
>>> x[np.ix_(rows, columns)]
array([[ 0,  2],
       [ 9, 11]])
```

Note that without the `np.ix_` call, only the diagonal elements would be selected:

```
>>> x[rows, columns]
array([ 0, 11])
```

This difference is the most important thing to remember about indexing with multiple advanced indices.

Example

A real-life example of where advanced indexing 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.

Boolean array indexing

This advanced indexing occurs when *obj* is an array object of Boolean type, such as may be returned from comparison operators. A single boolean index array is practically identical to

`x[obj.nonzero()]` where, as described above, `obj.nonzero()` returns a tuple (of length `obj.ndim`) of integer index arrays showing the `True` elements of *obj*. However, it is faster when `obj.shape == x.shape`.

If `obj.ndim == x.ndim`, `x[obj]` returns a 1-dimensional array filled with the elements of *x* corresponding to the `True` values of *obj*. The search order will be row-major, C-style. An index error will be raised if the shape of *obj* does not match the corresponding dimensions of *x*, regardless of whether those values are `True` or `False`.

A common use case for this is filtering for desired element values. For example, one may wish to select all entries from an array which are not `numpy.nan`:

```
>>> x = np.array([[1., 2.], [np.nan, 3.], [np.nan, np.nan]])
>>> x[~np.isnan(x)]
array([1., 2., 3.])
```

Or wish to add a constant to all negative elements:

```
>>> x = np.array([1., -1., -2., 3])
>>> x[x < 0] += 20
>>> x
array([ 1., 19., 18., 3.])
```

In general if an index includes a Boolean array, the result will be identical to inserting `obj.nonzero()` into the same position and using the integer array indexing mechanism described above. `x[ind_1, boolean_array, ind_2]` is equivalent to `x[(ind_1,) + boolean_array.nonzero() + (ind_2,)]`.

If there is only one Boolean array and no integer indexing array present, this is straightforward. Care must only be taken to make sure that the boolean index has *exactly* as many dimensions as it is supposed to work with.

In general, when the boolean array has fewer dimensions than the array being indexed, this is equivalent to `x[b, ...]`, which means x is indexed by b followed by as many `:` as are needed to fill out the rank of x. 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:

```
>>> x = np.arange(35).reshape(5, 7)
>>> b = x > 20
>>> b[:, 5]
array([False, False, False,  True,  True])
>>> x[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.

Example

From an array, select all rows which sum up to less or equal two:

```
>>> x = np.array([[0, 1], [1, 1], [2, 2]])
>>> rowsum = x.sum(-1)
>>> x[rowsum <= 2, :]
array([[0, 1],
       [1, 1]])
```

Combining multiple Boolean indexing arrays or a Boolean with an integer indexing array can best be understood with the `obj.nonzero()` analogy. The function `ix_` also supports boolean arrays and will work without any surprises.

Example

Use boolean indexing to select all rows adding up to an even number. At the same time columns 0 and 2 should be selected with an advanced integer index. Using the `ix_` function this can be done with:

```
>>> x = np.array([[ 0,  1,  2],
...               [ 3,  4,  5],
...               [ 6,  7,  8],
...               [ 9, 10, 11]])
>>> rows = (x.sum(-1) % 2) == 0
>>> rows
array([False,  True, False,  True])
>>> columns = [0, 2]
>>> x[np.ix_(rows, columns)]
array([[ 3,  5],
       [ 9, 11]])
```

Without the `np.ix_` call, only the diagonal elements would be selected.

Or without `np.ix_` (compare the integer array examples):

```
>>> rows = rows.nonzero()[0]
>>> x[rows[:, np.newaxis], columns]
array([[ 3,  5],
       [ 9, 11]])
```

Example

Use 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]])
```

Combining advanced and basic indexing

When there is at least one slice (`:`), ellipsis (`...`) or `newaxis` in the index (or the array has more dimensions than there are advanced indices), then the behaviour can be more complicated. It is like concatenating the indexing result for each advanced index element.

In the simplest case, there is only a *single* advanced index combined with a slice. For example:

```
>>> y = np.arange(35).reshape(5,7)
>>> y[np.array([0, 2, 4]), 1:3]
array([[ 1,  2],
       [15, 16],
       [29, 30]])
```

In effect, the slice and index array operation are independent. The slice operation extracts columns with index 1 and 2, (i.e. the 2nd and 3rd columns), followed by the index array operation which extracts rows with index 0, 2 and 4 (i.e the first, third and fifth rows). This is equivalent to:

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

A single advanced index can, for example, replace a slice and the result array will be the same. However, it is a copy and may have a different memory layout. A slice is preferable when it is possible. For example:

```
>>> x = np.array([[ 0,  1,  2],
...               [ 3,  4,  5],
...               [ 6,  7,  8],
...               [ 9, 10, 11]])
>>> x[1:2, 1:3]
array([[4, 5]])
>>> x[1:2, [1, 2]]
array([[4, 5]])
```

The easiest way to understand a combination of *multiple* advanced indices may be to think in terms of the resulting shape. There are two parts to the indexing operation, the subspace defined by the basic indexing (excluding integers) and the subspace from the advanced indexing part. Two cases of index combination need to be distinguished:

- The advanced indices are separated by a slice, `Ellipsis` or `newaxis`. For example `x[arr1, :, arr2]`.
- The advanced indices are all next to each other. For example `x[..., arr1, arr2, :]` but *not* `x[arr1, :, 1]` since `1` is an advanced index in this regard.

In the first case, the dimensions resulting from the advanced indexing operation come first in the result array, and the subspace dimensions after that. In the second case, the dimensions from the advanced indexing operations are inserted into the result array at the same spot as they were in the initial array (the latter logic is what makes simple advanced indexing behave just like slicing).

Example

Suppose `x.shape` is (10, 20, 30) and `ind` is a (2, 5, 2)-shaped indexing **`intp`** array, then `result = x[..., ind, :]` has shape (10, 2, 5, 2, 30) because the (20,)-shaped subspace has been replaced with a (2, 5, 2)-shaped broadcasted indexing subspace. If we let *i, j, k* loop over the (2, 5, 2)-shaped subspace then `result[..., i, j, k, :] = x[..., ind[i, j, k], :]`. This example produces the same result as **`x.take(ind, axis=-2)`**.

Example

Let `x.shape` be (10, 20, 30, 40, 50) and suppose `ind_1` and `ind_2` can be broadcast to the shape (2, 3, 4). Then `x[:, ind_1, ind_2]` has shape (10, 2, 3, 4, 40, 50) because the (20, 30)-shaped subspace from X has been replaced with the (2, 3, 4) subspace from the indices. However, `x[:, ind_1, :, ind_2]` has shape (2, 3, 4, 10, 30, 50) because there is no unambiguous place to drop in the indexing subspace, thus it is tacked-on to the beginning. It is always possible to use **`.transpose()`** to move the subspace anywhere desired. Note that this example cannot be replicated using **`take`**.

Example

Slicing can be combined with broadcasted boolean indices:

```
>>> x = np.arange(35).reshape(5, 7)
>>> b = x > 20
>>> b
array([[False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False],
       [ True,  True,  True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True,  True,  True]])
>>> x[b[:, 5], 1:3]
array([[22, 23],
       [29, 30]])
```

Field access

See also

Structured arrays

If the `ndarray` object is a structured array the fields of the array can be accessed by indexing the array with strings, dictionary-like.

Indexing `x['field-name']` returns a new view to the array, which is of the same shape as *x* (except when the field is a sub-array) but of data type `x.dtype['field-name']` and contains only the part of the data in the specified field. Also, record array scalars can be "indexed" this way.

Indexing into a structured array can also be done with a list of field names, e.g. `x[['field-name1', 'field-name2']]`. As of NumPy 1.16, this returns a view containing only those fields. In older versions of NumPy, it returned a copy. See the user guide section on Structured arrays for more information on multifield indexing.

If the accessed field is a sub-array, the dimensions of the sub-array are appended to the shape of the result. For example:

```
>>> x = np.zeros((2, 2), dtype=[('a', np.int32), ('b', np.float64, (3, 3))])
>>> x['a'].shape
(2, 2)
>>> x['a'].dtype
dtype('int32')
>>> x['b'].shape
(2, 2, 3, 3)
>>> x['b'].dtype
dtype('float64')
```

Flat iterator indexing

`x.flat` returns an iterator that will iterate over the entire array (in C-contiguous style with the last index varying the fastest). This iterator object can also be indexed using basic slicing or advanced indexing as long as the selection object is not a tuple. This should be clear from the fact that `x.flat` is a 1-dimensional view. It can be used for integer indexing with 1-dimensional C-style-flat indices. The shape of any returned array is therefore the shape of the integer indexing object.

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
Traceback (most recent call last):
  ...
TypeError: can't convert complex to int
```

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 that 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 indexing 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:

```
>>> z = np.arange(81).reshape(3, 3, 3, 3)
>>> 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 of 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
```

Detailed notes

These are some detailed notes, which are not of importance for day to day indexing (in no particular order):

- The native NumPy indexing type is `intp` and may differ from the default integer array type. `intp` is the smallest data type sufficient to safely index any array; for advanced indexing it may be faster than other types.
- For advanced assignments, there is in general no guarantee for the iteration order. This means that if an element is set more than once, it is not possible to predict the final result.
- An empty (tuple) index is a full scalar index into a zero-dimensional array. `x[()]` returns a *scalar* if `x` is zero-dimensional and a view otherwise. On the other hand, `x[...]` always returns a view.
- If a zero-dimensional array is present in the index *and* it is a full integer index the result will