Data Wrangling

Elements from NumPy arrays can be selected using four methods: scalar selection, slicing, numerical (or list-of-locations) indexing and logical (or Boolean) indexing.

Numerical Indexing

Numerical indexing uses lists or arrays of locations to select elements while logical indexing uses arrays containing Boolean values to select elements.

Numerical indexing, also called list-of-location indexing, is an alternative to slice notation. The fundamental idea underlying numerical indexing is to use coordinates to select elements, which is similar to the underlying idea behind slicing.

A numerical index can be either a list or a NumPy array and must contain integer data.

```
>>> x = 10 * arange(5.0)
>>> x[[0]] # List with 1 element
array([ 0.])
>>> x[[0,2,1]] # List
array([ 0., 20., 10.])
>>> sel = array([4,2,3,1,4,4]) # Array with repetition
>>>
```

```
>>> x[sel]
array([ 40., 20., 30., 10., 40., 40.])
>>> sel = array([[4,2],[3,1]]) # 2 by 2 array
>>> x[sel] # Selection has same size as sel
array([[ 40., 20.],
[ 30., 10.]])
>>> sel = array([0.0,1]) # Floating point data
>>> x[sel] # Error
IndexError: arrays used as indices must be of integer (or
>>> x[sel.astype(int)] # No error
array([ 10., 20.])
>>> x[0] # Scalar selection, not numerical indexing
1.0
```

```
>>> x = reshape(arange(10.0), (2,5))
>>> x
array([[ 0., 1., 2., 3., 4.],
[ 5., 6., 7., 8., 9.]])
>>> sel = array([0,1])
>>> x[sel,sel] # 1-dim arrays, no broadcasting
array([ 0., 6.])
>>> x[sel, sel+1]
array([ 1., 7.])
>>> sel_row = array([[0,0],[1,1]])
>>> sel_col = array([[0,1],[0,1]])
>>> x[sel_row,sel_col] # 2 by 2, no broadcasting
array([[ 0., 1.],
[ 5., 6.]])
```

```
>>> sel_row = array([[0],[1]])
>>> sel_col = array([[0,1]])
>>> # 2 by 1 and 1 by 2 - difference shapes, broadcasted
>>> x[sel_row,sel_col]
array([[ 0., 1.],
[ 5., 6.]])
```

Mixing Numerical Indexing with Scalar Selection

NumPy permits using difference types of indexing in the same expression. Mixing numerical indexing with scalar selection is trivial since any scalar can be broadcast to any array shape.

```
>>> x = array([[1,2],[3,4]])
>>> sel = x <= 3
>>> indices = nonzero(sel)
>>> indices
(array([0, 0, 1], dtype=int64), array([0, 1, 0], dtype=in
```

Mixing Numerical Indexing with Slicing

Mixing numerical indexing and slicing allow for entire rows or columns to be selected.

```
>>> x[:,[1]]
array([[ 2.],
[ 7.]])
>>> x[[1],:]
array([[ 6., 7., 8., 9., 10.]])
```

Mixing Numerical Indexing with Slicing

Note that the mixed numerical indexing and slicing uses a list ([1]) so that it is not a scalar. This is important since using a scalar will result in dimension reduction.

```
>>> x[:,1] # 1dimensional
array([ 2., 7.])
```

Numerical indexing and slicing can be mixed in more than 2-dimensions, although some care is required. In the simplest case where only one numerical index is used which is 1-dimensional, then the selection is equivalent to calling `ix_` where the slice a:b:s is replaced with arange(a,b,s).

```
>>> x = reshape(arange(3**3), (3,3,3)) # 3d
array
>>> sel1 = x[::2,[1,0],:1]
>>> sel2 = x[ix_(arange(0,3,2),[1,0],arange(0,1))]
>>> sel1.shape
(2L, 2L, 1L)
>>> sel2.shape
(2L, 2L, 1L)
>>> amax(abs(sel1sel2))
0
```

*

Linear Numerical Indexing using flat
Like slicing, numerical indexing can be combined with flat to select elements from an array using the row-major ordering of the array. The behavior of numerical indexing with flat is identical to that of using numerical indexing on a flattened version of the underlying array.

*

Linear Numerical Indexing using flat

```
>>> x.flat[[3,4,9]]
array([ 4., 5., 10.])
>>> x.flat[[[3,4,9],[1,5,3]]]
array([[ 4., 5., 10.],
[ 2., 6., 4.]])
```

Logical Indexing

Logical indexing differs fromslicing and numeric indexing by using logical indices to select elements, rows or columns. Logical indices act as light switches and are either on (True) or off (False). Pure logical indexing uses a logical indexing array with the same size as the array being used for selection and always returns a 1-dimensional array.

```
>>> x = arange(3,3)
>>> x < 0
array([ True, True, True, False, False, False], dtype=boo
>>> x[x < 0]
array([3,
2,
1])
>>> x[abs(x) >= 2]
array([3,
2,
2])
```

```
>>> x = reshape(arange(8,
8), (4,4))
>>> x[x < 0]
array([8,
7,
6,
5,
4,
3,
2,
1])
```

Logical Indexing

It is tempting to use two 1-dimensional logical arrays to act as row and column masks on a 2-dimensional array. This does not work, and it is necessary to use `ix_` if interested in this type of indexing.

```
>>> x = reshape(arange(8,8),(
4,4))
>>> cols = any(x < 6,
0)
>>> rows = any(x < 0, 1)
>>> cols
array([ True, True, False, False], dtype=bool
>>> rows
array([ True, True, False, False], dtype=bool)
>>> x[cols,rows] # Not upper 2 by 2
array([8,
3])
```

Logical Indexing

The difference between the final 2 commands is due to how logical indexing operates when more than logical array is used. When using 2 or more logical indices, they are first transformed to numerical indices using nonzero which returns the locations of the non-zero elements (which correspond to the True elements of a Boolean array).

*

Logical Indexing

```
>>> cols.nonzero()
(array([0, 1], dtype=int64),)
>>> rows.nonzero()
(array([0, 1], dtype=int64),)
```

Logical Indexing

The corresponding numerical index arrays have compatible sizes both are 2-element, 1-dimensional arrays and so numeric selection is possible. Attempting to use two logical index arrays which have non-broadcastable dimensions produces the same error as using two numerical index arrays with nonbroadcastable sizes.

Logical Indexing

```
>>> cols = any(x < 6,
0)
>>> rows = any(x < 4, 1)
>>> rows
array([ True, True, True, False], dtype=bool)
>>> x[cols,rows] # Error
ValueError: shape mismatch: objects cannot be broadcast t
```

`argwhere`

- `argwhere` returns an array containing the locations of elements where a logical condition is True.
- It is the same as `transpose(nonzero(x))`

```
>>> x = randn(3)
>>> x
array([-0.5910316 , 0.51475905, 0.68231135])
>>> argwhere(x<0.6)
array([[0],
[1]], dtype=int64)
```

```
>>> argwhere(x<-10.0) # Empty array
array([], shape=(0L, 1L), dtype=int64)
>>>
>>> x = randn(3,2)
>>> x
array([[ 0.72945913, 1.2135989 ],
[ 0.74005449, -1.60231553],
[ 0.16862077, 1.0589899 ]])
>>>
```

```
>>> argwhere(x<0)
array([[1, 1]], dtype=int64)
>>>
>>> argwhere(x<1)
array([[0, 0],
[1, 0],
[1, 1],
[2, 0]], dtype=int64)
```