NumPy Basics: Arrays and Vectorized Computation

Part 4

The NumPy ndarray: A Multidimensional Array Object

Part 4

Fancy Indexing

- *Fancy indexing* is a term adopted by NumPy to describe indexing using integer arrays.
- Suppose we had an 8 × 4 array:

```
In [98]: arr = np.empty((8, 4))
         for i in range(8):
             arr[i] = i
         arr

Out[98]: array([[0., 0., 0., 0.],
                [1., 1., 1., 1.],
                [2., 2., 2., 2.],
                [3., 3., 3., 3.],
                [4., 4., 4., 4.],
                [5., 5., 5., 5.],
                [6., 6., 6., 6.],
                [7., 7., 7., 7.]])
```

- To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

```
In [98]: arr
Out[98]: array([[0., 0., 0., 0.],
                [1., 1., 1., 1.],
                [2., 2., 2., 2.],
                [3., 3., 3., 3.],
                [4., 4., 4., 4.],
                [5., 5., 5., 5.],
                [6., 6., 6., 6.],
                [7., 7., 7., 7.]])

In [99]: arr[[4, 3, 0, 6]]
Out[99]: array([[4., 4., 4., 4.],
                [3., 3., 3., 3.],
                [0., 0., 0., 0.],
                [6., 6., 6., 6.]])
```

- Using negative indices selects rows from the end:

```
In [100]: arr

Out[100]: array([[0., 0., 0., 0.],
                 [1., 1., 1., 1.],
                 [2., 2., 2., 2.],
                 [3., 3., 3., 3.],
                 [4., 4., 4., 4.],
                 [5., 5., 5., 5.],
                 [6., 6., 6., 6.],
                 [7., 7., 7., 7.]])

In [101]: arr[[-3, -5, -7]]

Out[101]: array([[5., 5., 5., 5.],
                 [3., 3., 3., 3.],
                 [1., 1., 1., 1.]])
```

- Passing multiple index arrays does something slightly different; it selects a one-dimensional array of elements corresponding to each tuple of indices:

```
In [102]: arr = np.arange(32).reshape((8, 4))
          arr

Out[102]: 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]])

In [103]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]

Out[103]: array([ 4, 23, 29, 10])
```

- Here the elements (1, 0), (5, 3), (7, 1), and (2, 2) were selected.

- In order to obtain a rectangular region formed by selecting a subset of the matrix's rows and columns, here is one way to get that:

```
In [104]: arr
Out[104]: 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]])

In [105]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[105]: array([[ 4,  7,  5,  6],
                 [20, 23, 21, 22],
                 [28, 31, 29, 30],
                 [ 8, 11,  9, 10]])
```

- Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array.

Transposing Arrays and Swapping Axes

- Transposing is a special form of reshaping that similarly returns a view on the underlying data without copying anything.
- Arrays have the `transpose` method and also the special `T` attribute:

```
In [106]: arr = np.arange(15).reshape((3, 5))
          arr

Out[106]: array([[ 0,  1,  2,  3,  4],
                 [ 5,  6,  7,  8,  9],
                 [10, 11, 12, 13, 14]])

In [107]: arr.T

Out[107]: array([[ 0,  5, 10],
                 [ 1,  6, 11],
                 [ 2,  7, 12],
                 [ 3,  8, 13],
                 [ 4,  9, 14]])
```

- When doing matrix computations, you may do this very often—for example, when computing the inner matrix product using `np.dot`:

```
In [108]: arr = np.random.randn(6, 3)
          arr

Out[108]: array([[-0.06464487, -0.20582072, -0.07583151],
                 [ 0.22403978,  0.18865083, -0.10712158],
                 [-1.12435276, -0.34048034, -0.62308921],
                 [-1.54004002, -1.61862998, -0.20538688],
                 [-0.20561265, -2.25771857, -0.67447644],
                 [-0.40987589, -2.24229245, -0.79777542]])

In [109]: np.dot(arr.T, arr)

Out[109]: array([[ 3.90053999,  4.31442262,  1.4634485 ],
                 [ 4.31442262, 12.93900977,  3.85161789],
                 [ 1.4634485 ,  3.85161789,  1.53901347]])
```

- For higher dimensional arrays, `transpose` will accept a tuple of axis numbers to permute the axes:

```
In [110]: arr = np.arange(16).reshape((2, 2, 4))
          arr

Out[110]: array([[[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7]],

                 [[ 8,  9, 10, 11],
                  [12, 13, 14, 15]]])

In [111]: arr.transpose((1, 0, 2))

Out[111]: array([[[ 0,  1,  2,  3],
                  [ 8,  9, 10, 11]],

                 [[ 4,  5,  6,  7],
                  [12, 13, 14, 15]]])
```

- Here, the axes have been reordered with the second axis first, the first axis second, and the last axis unchanged.

- Simple transposing with `.T` is a special case of swapping axes.
- ndarray has the method `swapaxes`, which takes a pair of axis numbers and switches the indicated axes to rearrange the data:

```
In [112]: arr
Out[112]: array([[[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7]],

                 [[ 8,  9, 10, 11],
                  [12, 13, 14, 15]]])

In [113]: arr.swapaxes(1, 2)
Out[113]: array([[[ 0,  4],
                  [ 1,  5],
                  [ 2,  6],
                  [ 3,  7]],

                 [[ 8, 12],
                  [ 9, 13],
                  [10, 14],
                  [11, 15]]])
```

- `swapaxes` similarly returns a view on the data without making a copy.

Universal Functions: Fast Element-Wise Array Functions

- A universal function, or *ufunc*, is a function that performs element-wise operations on data in ndarrays.

- Many ufuncs are simple element-wise transformations, like `sqrt` or `exp`:

```
In [114]: arr = np.arange(10)
          arr

Out[114]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [115]: np.sqrt(arr)

Out[115]: array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
                 2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])

In [116]: np.exp(arr)

Out[116]: array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
                 5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
                 2.98095799e+03, 8.10308393e+03])
```

- These are referred to as *unary* ufuncs.
- Others, such as `add` or `maximum`, take two arrays (thus, *binary* ufuncs) and return a single array as the result:

```
In [117]: x = np.random.randn(8)
          y = np.random.randn(8)

In [118]: x
Out[118]: array([-1.1131831 ,  1.1305956 ,  1.52024064,  0.07550418,  0.39546478,
                 -1.00069004,  1.26966083,  0.93350173])

In [119]: y
Out[119]: array([ 1.14034354,  0.08557053,  1.74518133, -0.46595574, -1.79430514,
                 -0.10697579, -0.98502096, -0.84313744])

In [120]: np.maximum(x, y)
Out[120]: array([ 1.14034354,  1.1305956 ,  1.74518133,  0.07550418,  0.39546478,
                 -0.10697579,  1.26966083,  0.93350173])
```

- While not common, a ufunc can return multiple arrays.
- `modf` is one example, a vectorized version of the built-in Python `divmod`; it returns the fractional and integral parts of a floating-point array:

```
In [121]: arr = np.random.randn(7) * 5
          arr

Out[121]: array([-2.90625188, -1.92818667, -6.28527787, -1.9582203 ,  5.05081529,
                 -1.7765506 , -6.8635097 ])

In [122]: remainder, whole_part = np.modf(arr)

In [123]: remainder

Out[123]: array([-0.90625188, -0.92818667, -0.28527787, -0.9582203 ,  0.05081529,
                 -0.7765506 , -0.8635097 ])

In [124]: whole_part

Out[124]: array([-2., -1., -6., -1.,  5., -1., -6.])
```

- Ufuncs accept an optional out argument that allows them to operate in-place on arrays:

```
In [125]: arr
Out[125]: array([-2.90625188, -1.92818667, -6.28527787, -1.9582203 ,  5.05081529,
                 -1.7765506 , -6.8635097 ])

In [126]: np.sqrt(arr)
/home/joshua/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1: RuntimeWarning: invalid value encountered in
sqrt
  """Entry point for launching an IPython kernel.
Out[126]: array([      nan,       nan,       nan,       nan, 2.2474019,       nan,
                       nan])

In [127]: np.sqrt(arr, arr)
/home/joshua/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1: RuntimeWarning: invalid value encountered in
sqrt
  """Entry point for launching an IPython kernel.
Out[127]: array([      nan,       nan,       nan,       nan, 2.2474019,       nan,
                       nan])

In [128]: arr
Out[128]: array([      nan,       nan,       nan,       nan, 2.2474019,       nan,
                       nan])
```

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `arr ** 0.5`)
`square`	Compute the square of each element (equivalent to `arr ** 2`)
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
`ceil`	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)

`floor`	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as a separate array
`isnan`	Return boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-`NaN`) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise (equivalent to `~arr`).

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum; `fmax` ignores `NaN`
`minimum, fmin`	Element-wise minimum; `fmin` ignores `NaN`

`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding boolean array (equivalent to infix operators `>, >=, <, <=, ==, !=`)
`logical_and, logical_or, logical_xor`	Compute element-wise truth value of logical operation (equivalent to infix operators `& |, ^`)