NumPy Basics: Arrays and Vectorized Computation

Part 5

Array-Oriented Programming with Arrays

- Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops.
- This practice of replacing explicit loops with array expressions is commonly referred to as `vectorization`.
- In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations.

- As a simple example, suppose we wished to evaluate the function `sqrt(x^2 + y^2)` across a regular grid of values.
- The `np.meshgrid` function takes two 1D arrays and produces two 2D matrices corresponding to all pairs of `(x, y)` in the two arrays:

```
In [129]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points
          xs, ys = np.meshgrid(points, points)

In [130]: xs

Out[130]: array([[-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
                 [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
                 [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
                 ...,
                 [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
                 [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
                 [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99]])

In [131]: ys

Out[131]: array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
                 [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
                 [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
                 ...,
                 [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
                 [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
                 [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

- Now, evaluating the function is a matter of writing the same expression you would write with two points:

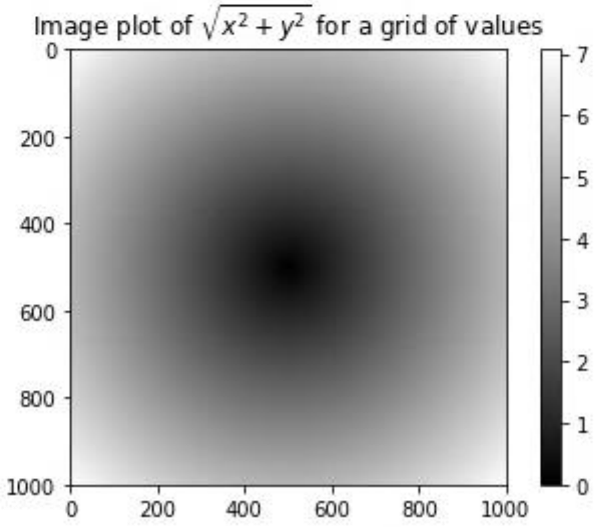
```
In [132]: z = np.sqrt(xs ** 2 + ys ** 2)
          z

Out[132]: array([[7.07106781, 7.06400028, 7.05693985, ..., 7.04988652, 7.05693985,
                  7.06400028],
                 [7.06400028, 7.05692568, 7.04985815, ..., 7.04279774, 7.04985815,
                  7.05692568],
                 [7.05693985, 7.04985815, 7.04278354, ..., 7.03571603, 7.04278354,
                  7.04985815],
                 ...,
                 [7.04988652, 7.04279774, 7.03571603, ..., 7.0286414 , 7.03571603,
                  7.04279774],
                 [7.05693985, 7.04985815, 7.04278354, ..., 7.03571603, 7.04278354,
                  7.04985815],
                 [7.06400028, 7.05692568, 7.04985815, ..., 7.04279774, 7.04985815,
                  7.05692568]])
```

- Now use matplotlib to create visualizations of this two-dimensional array:

```
In [135]: %matplotlib inline
          import matplotlib.pyplot as plt
          plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
          plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")

Out[135]: Text(0.5, 1.0, 'Image plot of $\\sqrt{x^2 + y^2}$ for a grid of values')
```



Expressing Conditional Logic as Array Operations

- The `numpy.where` function is a vectorized version of the ternary expression `x if condition else y`.
- Suppose we had a boolean array and two arrays of values:

```
In [137]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
          yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
          cond = np.array([True, False, True, True, False])
```

- Suppose we wanted to take a value from `xarr` whenever the corresponding value in `cond` is `True`, and otherwise take the value from `yarr`.
- A list comprehension doing this might look like:

```
In [138]: result = [(x if c else y)
                    for x, y, c in zip(xarr, yarr, cond)]
          result

Out[138]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

- This has multiple problems.
  - First, it will not be very fast for large arrays (because all the work is being done in interpreted Python code).
  - Second, it will not work with multidimensional arrays.
- With `np.where` you can write this very concisely:

```
In [139]: result = np.where(cond, xarr, yarr)
          result

Out[139]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

- The second and third arguments to `np.where` don't need to be arrays; one or both of them can be scalars.

- A typical use of `np.where` in data analysis is to produce a new array of values based on another array.
- Suppose you had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with −2.
- This is very easy to do with `np.where`:

```
In [140]: arr = np.random.randn(4, 4)
          arr

Out[140]: array([[-0.27849463,  0.46025818, -0.03859171,  0.03827386],
                 [ 2.37266443,  0.44533551,  0.13485372,  1.76568229],
                 [-0.90561492,  1.0540143 ,  0.54788435, -0.41625006],
                 [-1.19654663,  1.34126257, -1.25928655,  0.61601525]])

In [141]: arr > 0

Out[141]: array([[False,  True, False,  True],
                 [ True,  True,  True,  True],
                 [False,  True,  True, False],
                 [False,  True, False,  True]])

In [142]: np.where(arr > 0, 2, -2)

Out[142]: array([[-2,  2, -2,  2],
                 [ 2,  2,  2,  2],
                 [-2,  2,  2, -2],
                 [-2,  2, -2,  2]])
```

- You can combine scalars and arrays when using `np.where`.
- For example, I can replace all positive values in `arr` with the constant `2` like so:

```
In [143]: np.where(arr > 0, 2, arr) # set only positive values to 2

Out[143]: array([[-0.27849463,  2.        , -0.03859171,  2.        ],
                 [ 2.        ,  2.        ,  2.        ,  2.        ],
                 [-0.90561492,  2.        ,  2.        , -0.41625006],
                 [-1.19654663,  2.        , -1.25928655,  2.        ]])
```

Mathematical and Statistical Methods

- A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class.
- You can use aggregations (often called *reductions*) like `sum`, `mean`, and `std` (standard deviation) either by calling the array instance method or using the top-level NumPy function.

- Here I generate some normally distributed random data and compute some aggregate statistics:

```
In [144]: arr = np.random.randn(5, 4)
          arr

Out[144]: array([[-1.7161616 , -0.83950609,  0.2444356 , -1.28163466],
                 [-0.56114103, -1.26617331, -0.00615593,  0.80798954],
                 [-0.76547653, -0.80233489, -1.24336128,  0.2962289 ],
                 [ 0.70000169, -2.09674882,  0.5030099 , -0.95107228],
                 [-1.72963723,  1.18475635, -0.79401017, -0.73355954]])

In [145]: arr.mean()

Out[145]: -0.5525275694463618

In [146]: np.mean(arr)

Out[146]: -0.5525275694463618

In [147]: arr.sum()

Out[147]: -11.050551388927236
```

- Functions like `mean` and `sum` take an optional `axis` argument that computes the statistic over the given axis, resulting in an array with one fewer dimension:

```
In [148]: arr

Out[148]: array([[-1.7161616 , -0.83950609,  0.2444356 , -1.28163466],
                 [-0.56114103, -1.26617331, -0.00615593,  0.80798954],
                 [-0.76547653, -0.80233489, -1.24336128,  0.2962289 ],
                 [ 0.70000169, -2.09674882,  0.5030099 , -0.95107228],
                 [-1.72963723,  1.18475635, -0.79401017, -0.73355954]])

In [149]: arr.mean(axis=1)

Out[149]: array([-0.89821669, -0.25637018, -0.62873595, -0.46120238, -0.51811265])

In [150]: arr.sum(axis=0)

Out[150]: array([-4.0724147 , -3.82000677, -1.29608188, -1.86204804])
```

- The way to understand the *axis* of `mean` and `sum` is that it *collapses* the specified axis.
- So when it collapses the axis 0 (row), it becomes just one row and column-wise sum.

- Other methods like `cumsum` and `cumprod` do not aggregate, instead producing an array of the intermediate results:

```
In [151]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
          arr.cumsum()

Out[151]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

- In multidimensional arrays, accumulation functions like `cumsum` return an array of the same size, but with the partial aggregates computed along the indicated axis according to each lower dimensional slice:

```
In [152]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
          arr

Out[152]: array([[0, 1, 2],
                 [3, 4, 5],
                 [6, 7, 8]])

In [153]: arr.cumsum(axis=0)

Out[153]: array([[ 0,  1,  2],
                 [ 3,  5,  7],
                 [ 9, 12, 15]])

In [154]: arr.cumprod(axis=1)

Out[154]: array([[  0,   0,   0],
                 [  3,  12,  60],
                 [  6,  42, 336]])
```

Method	Description
`sum`	Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
`mean`	Arithmetic mean; zero-length arrays have `NaN` mean
`std, var`	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator `n`)
`min, max`	Minimum and maximum
`argmin, argmax`	Indices of minimum and maximum elements, respectively
`cumsum`	Cumulative sum of elements starting from 0
`cumprod`	Cumulative product of elements starting from 1

Methods for Boolean Arrays

- Boolean values are coerced to `1` (`True`) and `0` (`False`) in the preceding methods.
- Thus, `sum` is often used as a means of counting `True` values in a boolean array:

```
In [155]: arr = np.random.randn(100)
          (arr > 0).sum() # Number of positive values
Out[155]: 45
```

- There are two additional methods, `any` and `all`, useful especially for boolean arrays.
- `any` tests whether one or more values in an array is `True`, while `all` checks if every value is `True`:

```
In [156]: bools = np.array([False, False, True, False])

In [157]: bools.any()
Out[157]: True

In [158]: bools.all()
Out[158]: False
```

- These methods also work with non-boolean arrays, where non-zero elements evaluate to `True`.

Sorting

- Like Python's built-in list type, NumPy arrays can be sorted in-place with the `sort` method:

```
In [159]: arr = np.random.randn(6)

In [160]: arr
Out[160]: array([-0.49457911, -0.44779719,  0.43820231,  0.31435367, -1.02623222,
                 -0.34316151])

In [161]: arr.sort()

In [162]: arr
Out[162]: array([-1.02623222, -0.49457911, -0.44779719, -0.34316151,  0.31435367,
                  0.43820231])
```

- You can sort each one-dimensional section of values in a multidimensional array in-place along an axis by passing the axis number to `sort`:

```
In [163]: arr = np.random.randn(5, 3)

In [164]: arr

Out[164]: array([[ 0.30835203, -0.19957533,  1.39523675],
                 [ 1.61598882, -0.28583692,  0.10466761],
                 [ 1.14346706,  0.99471286,  0.59694469],
                 [ 1.70069953, -0.16645399, -0.99551559],
                 [ 0.08251799, -0.21560452,  1.30253489]])

In [165]: arr.sort(1)

In [166]: arr

Out[166]: array([[-0.19957533,  0.30835203,  1.39523675],
                 [-0.28583692,  0.10466761,  1.61598882],
                 [ 0.59694469,  0.99471286,  1.14346706],
                 [-0.99551559, -0.16645399,  1.70069953],
                 [-0.21560452,  0.08251799,  1.30253489]])
```

- The top-level method `np.sort` returns a sorted copy of an array instead of modifying the array in-place.

- A quick-and-dirty way to compute the quantiles of an array is to sort it and select the value at a particular rank:

```
In [167]: large_arr = np.random.randn(1000)
          large_arr.sort()
          large_arr[int(0.05 * len(large_arr))] # 5% quantile

Out[167]: -1.626018229951018
```

Unique and Other Set Logic

- NumPy has some basic set operations for one-dimensional ndarrays.
- A commonly used one is `np.unique`, which returns the sorted unique values in an array:

```
In [168]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
          np.unique(names)

Out[168]: array(['Bob', 'Joe', 'Will'], dtype='<U4')

In [169]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
          np.unique(ints)

Out[169]: array([1, 2, 3, 4])
```

- Another function, `np.in1d`, tests membership of the values in one array in another, returning a boolean array:

```
In [170]: values = np.array([6, 0, 0, 3, 2, 5, 6])
          np.in1d(values, [2, 3, 6])

Out[170]: array([ True, False, False,  True,  True, False,  True])
```

Method	Description
`unique(x)`	Compute the sorted, unique elements in `x`
`intersect1d(x, y)`	Compute the sorted, common elements in `x` and `y`
`union1d(x, y)`	Compute the sorted union of elements
`in1d(x, y)`	Compute a boolean array indicating whether each element of `x` is contained in `y`
`setdiff1d(x, y)`	Set difference, elements in `x` that are not in `y`
`setxor1d(x, y)`	Set symmetric differences; elements that are in either of the arrays, but not both