Scientific Computing in Python: A NumPy Crash-Course Cluedump

Mark Chilenski
(markchil)

January 25, 2022

You can download the code to follow along at:
https://github.com/markchil/numpy-lecture

Intro

Goals for today:

- Introduce the core elements of NumPy
- Show how to write clean, fast code using NumPy

Prerequisites:

- Basic familiarity with Python

You can download the code to follow along at:
https://github.com/markchil/numpy-lecture

Scientific Computing in Python: A NumPy Crash-Course Cluedump

Scientific Computing in Python: A NumPy Crash-Course Cluedump

What Is NumPy?

- NumPy is the main Python package for performing numerical computations
- NumPy provides data structures which enable efficient manipulation of arrays of data with expressive syntax

Motivating Example: Evaluating $z = ax^2 + by^2 + cxy$ Over a 2d Grid

Without NumPy, setting up the x and y grids is pretty ugly:

```
x = [2.0 * v / (200 - 1) - 1.0 for v in range(200)]
y = [2.0 * v / (201 - 1) - 1.0 for v in range(201)]
```

Evaluating the function itself requires nested for-loops:

```
def quadratic_pure_python(x, y):
    z = []
    for x_val in x:
        row = []
        for y_val in y:
            row.append(a * x_val ** 2 + b * y_val
                ** 2 + c * x_val * y_val)
        z.append(row)
    return z
```

Evaluating $z = ax^2 + by^2 + cxy$ Over a 2d Grid With NumPy

First, we import NumPy:

```
1 import numpy as np
```

Defining our grids is easy with `np.linspace`:

```
2 x = np.linspace(-1.0, 1.0, 200)
3 y = np.linspace(-1.0, 1.0, 201)
```

And evaluating the function itself requires no loops:

```
4 def quadratic_numpy():
5     z = (
6         a * x[:, np.newaxis] ** 2 + b * y[np.
              newaxis, :] ** 2 +
7         c * x[:, np.newaxis] * y[np.newaxis, :]
8     )
9     return z
```

Performance Comparison

Pure Python:	14.2 ms
NumPy:	138 µs

- The NumPy solution is not only cleaner (no nested for-loops), it is *100× faster*
- NumPy obtains this speed by performing the majority of its operations in fast, compiled C/Fortran code: you write your code in Python, but NumPy does the heavy lifting in compiled libraries

Scientific Computing in Python: A NumPy Crash-Course Cluedump

The `ndarray` Class

- The `ndarray` class stores arrays of N-dimensional data
  - Usually just called "array" or "numpy array"
- A given array `x` consists of:
  - A chunk of memory
  - A datatype which indicates what type of value is stored in the memory (`x.dtype`)
  - A shape which indicates how the elements are arranged in an N-dimensional grid (`x.shape`)

Scientific Computing in Python: A NumPy Crash-Course Cluedump

Creating Arrays From Pure Python Data Structures

- Create an array from an existing list:

```
x = np.asarray([1, 2, 3, 4], dtype=float)
print(x)
# [1. 2. 3. 4.]
```

  - NumPy tries to guess the `dtype` based on the input: need to specify `dtype=float` here to avoid defaulting to `int` based on the input

- Create a 2d array using nested lists:

```
x = np.asarray([[1, 2, 3], [4, 5, 6]], dtype=
    float)
print(x)
"""
[[1. 2. 3.]
 [4. 5. 6.]]
"""
```

Creating Arrays With Specified Shapes

- Create an array of zeros:

```
x = np.zeros((3, 2))
print(x)
"""
[[0. 0.]
 [0. 0.]
 [0. 0.]]
"""
```

- Create an array of ones:

```
x = np.ones(5)
print(x)
# [1. 1. 1. 1. 1.]
```

Creating Grids of Numbers

- We already saw how to create linearly-spaced numbers:

```
x = np.linspace(0, 1, 4)
print(x)
# [0.         0.33333333 0.66666667 1.        ]
```

- We can also create logarithmically-spaced numbers:

```
x = np.logspace(-1, 1, 5)
print(x)
# [ 0.1         0.31622777  1.
    3.16227766 10.        ]
```

  - You specify the *exponent* of the starting and ending points
  - The default base is 10, you can change this with the `base` keyword

Indexing Into 1d Arrays

You can index into 1d arrays just like with Python lists:

```
x = np.linspace(0, 1, 5)
print(x)
# [0.   0.25 0.5  0.75 1.  ]
```

- Indices start from zero:

```
print(x[0])
# 0.0
```

- Negative indexing goes from the end of the array, with -1 being the last element:

```
print(x[-1])
# 1.0
```

Slice Indexing

```
x = np.linspace(0, 1, 5)
print(x)
# [0.   0.25 0.5  0.75 1.  ]
```

You can access a range of elements using a slice:

```
print(x[1:3])
# [0.25 0.5 ]
```

- The number before the colon is the element the slice starts from
- The number after the colon is *one more than* the element the slice stops at

Slice Indexing

```
x = np.linspace(0, 1, 5)
print(x)
# [0.   0.25 0.5  0.75 1.  ]
```

- You can specify a stride for the slice:

```
print(x[1:4:2])
# [0.25 0.75]
```

- As a shorthand, you can omit the start index (it will default to 0) and/or the stop index (it will default to `len(x)`):

```
print(x[::2])
# [0.  0.5 1. ]
print(x[1::2])
# [0.25 0.75]
```

Exacting Elements at Specific Indices

```
x = np.linspace(0, 1, 5)
print(x)
# [0.   0.25 0.5  0.75 1.  ]
```

You can use an array (or list) of integers to extract the elements at specific values:

```
print(x[[0, 2, 3]])
# [0.   0.5  0.75]
```

Boolean Indexing

```
x = np.linspace(0, 1, 5)
print(x)
# [0.   0.25 0.5  0.75 1.  ]
```

- You can use an array (or list) of booleans with the same length as the array to extract specific elements:

```
print(x[[True, False, True, True, False]])
# [0.   0.5  0.75]
```

- The use of boolean array expressions for indexing is extremely powerful:

```
print(x[(x >= 0.5) & (x < 1.0)])
# [0.5  0.75]
```

Indexing Into 2d Arrays

You can provide a comma-separated list of any of the types of indexing shown above to extract elements from a 2-dimensional array:

```
x = np.random.randn(3, 4)
print(x)
"""
[[ 0.04982776 -1.12826248 -0.08939734 -0.74356624]
 [-1.42371767  0.72748056  0.38503782 -1.72328503]
 [-1.61074892 -1.17030821 -0.84875467 -0.72356627]]
"""
```

The first index is the row index, the second index is the column index:

```
print(x[0, 1])
# -1.128262478701515
```

Indexing Into 2d Arrays

```
x = np.random.randn(3, 4)
print(x)
"""
[[ 0.04982776 -1.12826248 -0.08939734 -0.74356624]
 [-1.42371767  0.72748056  0.38503782 -1.72328503]
 [-1.61074892 -1.17030821 -0.84875467 -0.72356627]]
"""
```

To get all of the entries along a given axis (e.g., an entire row), use the slice operator `:` with no start, end, or stride indicated:

```
print(x[0, :])
# [ 0.04982776 -1.12826248 -0.08939734 -0.74356624]
```

You can omit the trailing `:` when accessing entire rows:

```
print(x[0])
# [ 0.04982776 -1.12826248 -0.08939734 -0.74356624]
```

Indexing Into Higher-Dimensional Arrays

Pretty much the same as for 2d: provide a comma-separated list of indices, one for each dimension

```
x = np.random.randn(2, 3, 4)
```

You can omit any number of trailing :'s:

```
print(x[0, :, :])
"""
[[-0.1221692   0.53916002  0.9522174  -0.4032439 ]
 [-0.30484904 -1.74816515  1.00955669  1.68279295]
 [ 0.8801445  -0.68018485 -0.20127799 -1.25986568]]
"""
print(x[0])
"""
[[-0.1221692   0.53916002  0.9522174  -0.4032439 ]
 [-0.30484904 -1.74816515  1.00955669  1.68279295]
 [ 0.8801445  -0.68018485 -0.20127799 -1.25986568]]
"""
```

Indexing Into Higher-Dimensional Arrays

```
x = np.random.randn(2, 3, 4)
```

You can omit any number of leading :'s by using ...:

```
print(x[:, :, 0])
"""
[[-0.1221692  -0.30484904  0.8801445 ]
 [ 0.53008465 -1.72623285  0.71619319]]
"""
print(x[..., 0])
"""
[[-0.1221692  -0.30484904  0.8801445 ]
 [ 0.53008465 -1.72623285  0.71619319]]
"""
```

Adding New Dimensions With Indexing

You can add a new dimension (with size 1) to an array by including `np.newaxis` in the index expression:

```
x = np.random.randn(4, 3)
y = x[:, np.newaxis, :]
print(x.shape)
# (4, 3)
print(y.shape)
# (4, 1, 3)
```

This will be very useful when setting up certain operations

Scientific Computing in Python: A NumPy Crash-Course Cluedump

Vectorization: The Key to Speed

- Almost all NumPy operations are *vectorized*: a single function call causes the same operation to be applied on every element of an array
- This avoids the need to write loops in Python: instead, you let NumPy do the loop in its fast, compiled libraries
- Code which avoids loops will (almost) always be faster than code with loops, and is usually cleaner/easier to understand

Speed Test: Exponentiation

Let's try squaring a bunch of numbers:

```
x = np.linspace(0, 1, 5000)
```

First we do it with a loop:

```
def square_loop(x):
    y = np.zeros_like(x)
    for idx, x_val in enumerate(x):
        y[idx] = x_val ** 2
    return y
```

Then with a single line using NumPy's vectorized exponentiation operation:

```
def square_numpy(x):
    return x ** 2
```

`square_loop()` takes 3 ms, `square_numpy()` takes 5.0 µs: *the vectorized version is 600× faster*

Unary Operations

Unary operations are applied to each element:

```
x = np.linspace(0, 1, 5)
print(x)
# [0.   0.25 0.5  0.75 1.  ]
```

Scalar addition:

```
y = x + 10
print(y)
# [10.   10.25 10.5  10.75 11.  ]
```

Scalar multiplication:

```
y = 5 * x
print(y)
# [0.   1.25 2.5  3.75 5.  ]
```

Scalar division:

```
y = x / 2
print(y)
# [0.    0.125 0.25  0.375 0.5  ]
```

Unary Operations

```
x = np.linspace(0, 1, 5)
print(x)
# [0.   0.25 0.5  0.75 1.  ]
```

Negation:

```
y = -x
print(y)
# [-0.   -0.25 -0.5  -0.75 -1.  ]
```

Exponentiation:

```
y = x ** 2
print(y)
# [0.     0.0625 0.25   0.5625 1.    ]
```

Unary Operations

Some operations are provided as functions instead of operators:

```
x = np.linspace(0, 1, 5)
print(x)
# [0.   0.25 0.5  0.75 1.  ]
```

Square root:

```
y = np.sqrt(x)
print(y)
# [0.         0.5        0.70710678 0.8660254  1.
           ]
```

Exponential ($y = e^x$):

```
y = np.exp(x)
print(y)
# [1.         1.28402542 1.64872127 2.11700002
    2.71828183]
```

Unary Operations

Some operations are provided as functions instead of operators:

```
x = np.linspace(0, 1, 5)
print(x)
# [0.   0.25 0.5  0.75 1.  ]
```

(Natural) logarithm (note the warning from taking log(0)):

```
y = np.log(x)
# <string>:507: RuntimeWarning: divide by zero
    encountered in log
print(y)
# [       -inf -1.38629436 -0.69314718 -0.28768207
    0.        ]
```

(Trigonometric) sine:

```
y = np.sin(x)
print(y)
# [0.         0.24740396 0.47942554 0.68163876
    0.84147098]
```

Binary Operations

- Binary operations act element-by-element
- In their most basic form, this means that the two operands must have the same shape
    - But we'll see an exception to this when we get to broadcasting

Examples of Binary Operations

```
x = np.linspace(0, 1, 5)
y = np.linspace(1, 2, 5)
print(x)
# [0.   0.25 0.5  0.75 1.  ]
print(y)
# [1.   1.25 1.5  1.75 2.  ]
```

Addition:

```
z = x + y
print(z)
# [1.  1.5 2.  2.5 3. ]
```

Division (note warning on divide-by-zero):

```
z = y / x
# <string>:557: RuntimeWarning: divide by zero
    encountered in true_divide
print(z)
# [       inf 5.          3.          2.33333333 2.
          ]
```

Broadcasting: Binary Operations on Differently-Sized Arrays

- Broadcasting is a powerful syntax for efficiently performing calculations on "tensor product" grids of inputs
- It is a very useful tool for removing the need to use loops (and hence dramatically speeding up your code)

Basic Broadcasting Example: $z = xy$

Let's try to compute $z = xy$ over the grid $-1 \leq x \leq 0$, $0 \leq y \leq 1$:

```
x = np.linspace(-1, 0, 5)
y = np.linspace(0, 1, 6)
```

We can't directly multiply `x` and `y` because they have different shapes:

```
    z = x * y
```

```
Traceback (most recent call last):
  File "<string>", line 590, in <module>
ValueError: operands could not be broadcast
    together with shapes (5,) (6,)
```

Basic Broadcasting Example: $z = xy$

```
x = np.linspace(-1, 0, 5)
y = np.linspace(0, 1, 6)
```

We can use `np.newaxis` to reshape `x` and `y`:

```
x_reshaped = x[:, np.newaxis]
y_reshaped = y[np.newaxis, :]
print(x_reshaped.shape)
# (5, 1)
print(y_reshaped.shape)
# (1, 6)
```

And this allows us to compute the desired expression:

```
z = x_reshaped * y_reshaped
print(z.shape)
# (5, 6)
```

What Just Happened? Full Details of Broadcasting

Two arrays can be *broadcast* together if their shapes are *compatible* according to the following rules:

1. If one array has fewer dimensions than the other, add 1's at the *beginning* of the lower-dimensional array's shape
2. For each element of the shapes, either:
    - The two arrays must have the same number of elements along that dimension, or
    - One (or both) of the arrays must have 1 element along that dimension

Whenever there is a 1 in one array's shape, NumPy pretends that the array has been repeated along that axis however many times are necessary to match the shape of the other array

Example of Broadcasting

Whenever there is a 1 in one array's shape, NumPy pretends that the array has been repeated along that axis however many times are necessary to match the shape of the other array:

```
x = np.zeros((2, 1, 3))
y = np.zeros((5, 1, 4, 3))
z = x + y
print('   ', x.shape)
#     (2, 1, 3)
print(y.shape)
# (5, 1, 4, 3)
print(z.shape)
# (5, 2, 4, 3)
```

Aggregating Information Along Array Axes

```
x = np.linspace(0, 1, 5)
print(x)
# [0.   0.25 0.5  0.75 1.  ]
```

Sum of elements:

```
print(x.sum())
# 2.5
```

Minimum value:

```
print(x.min())
# 0.0
```

Maximum value:

```
print(x.max())
# 1.0
```

Mean value:

```
print(x.mean())
# 0.5
```

Aggregating in Multidimensional Arrays

On multidimensional arrays, the default behavior is to aggregate across all of the elements of the array to yield a scalar:

```
x = np.ones((3, 4, 5, 6))
print(x.sum())
# 360.0
```

Aggregating in Multidimensional Arrays

```
x = np.ones((3, 4, 5, 6))
```

You can specify an axis to do the aggregation on – the result will have one fewer dimension than the input:

```
s = x.sum(axis=2)
print(x.shape)
# (3, 4, 5, 6)
print(s.shape)
# (3, 4, 6)
```

Aggregating in Multidimensional Arrays

```
x = np.ones((3, 4, 5, 6))
```

You can specify also aggregate over multiple dimensions at once:

```
s = x.sum(axis=(1, 3))
print(x.shape)
# (3, 4, 5, 6)
print(s.shape)
# (3, 5)
```

Aggregating Without Removing Dimensions

Suppose we want to normalize each row of a matrix by its 2-norm:

```
x = 10 * np.random.randn(3, 4)
x_norm = np.linalg.norm(x, axis=1)
print(x.shape)
# (3, 4)
print('   ', x_norm.shape)
#     (3,)
```

`x` and `x_norm` are not compatible: we can't just take `x/x_norm`

Aggregating Without Removing Dimensions

Use the `keepdims` flag to ensure the resulting shapes are compatible:

```
x_norm = np.linalg.norm(x, axis=1, keepdims=True)
x_normalized = x / x_norm
print(x.shape)
# (3, 4)
print(x_norm.shape)
# (3, 1)
```

Scientific Computing in Python: A NumPy Crash-Course Cluedump

Doing Stuff With Arrays: The Scientific Python Ecosystem

Many packages use NumPy to move data around and do stuff, here are a few:

- NumPy (numpy.org) itself has subpackages for:
  - Linear algebra (`numpy.linalg`)
  - Taking discrete Fourier transforms (`numpy.fft`)
  - Generating random numbers (`numpy.random`)
- SciPy (scipy.org) was developed jointly with NumPy, and provides many high-level routines, including:
  - Numeric integration (including ODE solvers) (`scipy.integrate`)
  - Optimization (`scipy.optimize`)
  - Interpolation/smoothing (`scipy.interpolate`)
  - Statistics (`scipy.stats`)
- Matplotlib (matplotlib.org) provides a variety of plotting routines
- scikit-learn (scikit-learn.org) provides a huge range of machine learning algorithms and supporting tools