NumPy - Thinking in Arrays

June 2, 2016

NumPy

Motivation

At the core of most computational physics problems lives an **array**. Arrays are a natural way to describe numerical and discretized problems. Because geometry can be dissected into tetrahedrons (pyramids) or hexahedrons (cubes) and arrays can then be used to represent scalar or vector values of each point in three dimensional space. Operations on arrays can furthermore be used to represent or approximate calculus operations, such as integration or differentiation.
For a computer an array is just a contiguous block of memory where every element has the same type and layout.

NumPy

Motivation

Every programming language that is serious about scientific computing has a notion of a **array data language**. Some languages such as MATLAB or Mathematica are centered around the array data language. In Fortran arrays are first class citizens as well.

Python has NumPy!

NumPy is easy to learn, intuitive to use, and orders of magnitudes faster than using pure Python for array-based operations.

NumPy

Arrays

The basic data type that NumPy provides is the *n*-dimensional array class `ndarray`.
You can create an array in NumPy using the `array()` function.

```
In [1]: import numpy as np
In [2]: np.array([6, 28, 496, 8218])
Out[2]: array([   6,   28,  496, 8218])
```

There are various other ways besides `array()` to create arrays in NumPy. The four most common ones are `arange()`, `zeros()`, `ones()`, and `empty()`.

NumPy

Arrays

The `arange()` function works just like Python's `range()` function. It takes, `start`, `stop`, and `step` arguments and returns an `ndarray`.

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

The `zeros()` and `ones()` functions take an integer or a tuple of integers as parameter and return an `ndarray` whose shape matches that of the tuples and whose elements contain only zeros or ones.

```
np.zeros(4)
array([0., 0., 0., 0.])

np.ones((2, 3))
array([1., 1., 1.],
      [1., 1., 1.])
```

NumPy

Arrays

The `empty()` function will simply allocate memory, but will not assign any values!
This means that the contents of that array will be whatever happened to be in memory at the time. Often this looks like random noise, sometimes you might just get zeros, but that behavior is not guaranteed.
Empty arrays are most used if you have existing data, that you want to store in an array and you do not want to "waste" computing time by setting all values to zero first, if you are going to overwrite them anyways.

```
np.empty(4)
array([0., 0., 0., 0.]) # if you're lucky
np.empty(4)
array([6.93625398e-310, 6.93625398e-310,
       0.00000000e+000, 0.00000000e+000]) # more likely
```

NumPy

Arrays

Also useful functions to generate `ndarrays` are `linspace()` and `logspace()`. These create an even linearly- or logarithmically-spaced grid of points between a lower and upper bound that is inclusive on both ends. `logspace()` has a `base` keyword argument, that defaults to 10.

```
np.linspace(1, 2, 5)
array([1., 1.25, 1.5, 1.75, 2.])

np.logspace(1, -1, 3)
array([10., 1., 0.1])
```

NumPy

Array attributes

Under the cover `ndarray` is a Python object with a fixed block of memory and metadata that defines the features of that array. Here is a list of some of these attributes.

```
data      # Buffer to the raw array data
dtype     # data type of data
ndim      # number of dimensions
shape     # rank of dimensions
size      # total number of elements
itemsize  # number of bytes per element
nbytes    # total number of bytes
```

NumPy

Array attributes

Modifying the attributes in an allowable way automatically updates the values of the other attributes as well. Since the data buffer is fixed-length, all modifications must preserve the array size. This means you cannot append data to an array, without creating a new array.

A common manipulation is the reshaping of an array.

```
a = np.arange(4)
array([0, 1, 2, 3])

a.shape = (2,2)
array([0, 1],
      [2, 3])
```

NumPy also has the `reshape()` function that you can call with an array argument. But this will return a reshaped copy of the original array, thus doubling the used memory. The convention is as follows: Operations on attributes occur in-place, functions that take an array as argument create a copy.

NumPy

Data types

The `dtype` attribute is the most important one. It determines the size and meaning of each element of the array. NumPy, of course, heavily focusses on numeric types. All elements have to be of the same data type and have to have a constant size in memory. Even strings must have a fixed size in an array. This is so that the array as a whole has predictable properties.
NumPy has more data types than standard Python.
These include 8, 16, 32, and 64 bit versions of integers and unsigned integers, 8, 16, 32, 64, 96, and 128 bit versions of `float`.

NumPy

Data types

When creating an array, the `dtype` is automatically selected and will always be that of the least precise element. So if you have a list of integers and one single float, when you create an array from that list, the array data type would be `float64`, because floats are less precise that integers.

```
a = np.array([6, 26, 396, 8128])
array([6, 26, 496, 8128])
dtype('int64')

b = np.array([6, 26.0, 396, 8128])
array([  6.00000000e+00,   2.60000000e+01,   3.96000000e+02,
         8.12800000e+03])
dtype('float64')
```

The order of data types sorted from greatest to least precision is boolean, unsigned integer, integer, float, complex, string, and object.

NumPy

Data types

You can force an array to have a given data type by passing a `dtype=` as a keyword argument upon creation. This will convert all elements to that data type instead of automatically choosing the least precise one. In some cases this can cause a loss of information (`float` to `int` conversion, for instance). However, it has the benefit of giving you exactly what you want.
Providing an explicit `dtype` is a good idea in most cases, because it makes your code more readable.

```
a = np.array([6, 28.0, 496, 8128], dtype=np.int8)
array([  6,  28, -16, -64], dtype=int8) # overflow

b = np.array([6, 28.0, 496, 8128], dtype='float32')
array([  6.00000000e+00, 2.80000000e+01,
         4.96000000e+02, 8.12800000e+03], dtype=float32)
```

NumPy

Array slicing

NumPy arrays have the same slicing semantics of Python lists when accessing elements or sub-arrays.

```
a = np.arange(8)
array([0, 1, 2, 3, 4, 5, 6, 7])

a[::-1]
array([7, 6, 5, 4, 3, 2, 1, 0])

a[2:6]
array([2, 3, 4, 5])

a[1::3]
array([1, 4, 7])
```

NumPy

Array slicing

Since arrays in NumPy are n-dimensional you can slice along any and all axes. This is something that would be more complicated in standard Python using lists of lists.

```
# Python
outer = [...]
selection = [inner[a:b:c] for inner in outer[x:y:z]]

# NumPy
outer = np.array([...])
selection = outer[x:y:z, a:b:c]
```

The `for`-loops are implicitly handled by NumPy and its underlying C-layer. This make executing complex slices much faster in NumPy than in standard Python.

NumPy

Array slicing

Let us look at an example.

```
# Create a 1D array and reshape it to be 4x4
a = np.arange(16)
a.shape = (4, 4)

array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

# Slice the even rows and the odd columns
```

NumPy

Array slicing

Let us look at an example.

```
# Create a 1D array and reshape it to be 4x4
a = np.arange(16)
a.shape = (4, 4)

array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

# Slice the even rows and the odd columns

a[::2, 1::2]
array([[ 1,  3],
       [ 9, 11]])
```

NumPy

Array slicing

```
# our array
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

# Slice the inner 2x2 array
a[1:3, 1:3]
array([[ 5,  6],
       [ 9, 10]])

# reverse the first 3 rows taking the first 3 columns
a[2::-1, :3]
array([[ 8,  9, 10],
       [ 4,  5,  6],
       [ 0,  1,  2]])
```

NumPy

Array slicing

The most important thing to understand about array slicing, is that slices are **views** into the original array. No data is copied when creating a slice. This again makes working with array very fast in NumPy. If you create a slice of an array, that is itself a slice, both slices will point back to the original array. That also means if you manipulate a slice, the changes are reflected back in the original array.

```
a = np.arange(6)
array([0, 1, 2, 3, 4, 5])
b = a[1::2]
array([1, 3, 5])
b[1] = 42
array([1, 42, 5])
# a is then
array([0, 1, 2, 42, 4, 5])
```

NumPy

Array slicing

To see if an array is a view of another array you can check its base attribute. This will point to the original array, where the data is stored if it is a view. It is empty if the array data is stored in this object.

```
a = np.arange(6)
array([0, 1, 2, 3, 4, 5])
a.base
None

b = a[1::2]
b.base
array([0, 1, 2, 3, 4, 5])

b.base is a
True
```

NumPy

Array slicing

If you truly want a copy of a slice of an array, you can always create a new array from a slice using the type name `array`.

```
a = np.arange(16)
b = np.array(a[1::11])
```

Slices are not the only way of creating a view. The `ndarray` class has a `view()` method that will give you a view into the whole array. This method takes two arguments. The `dtype` keyword argument allows you to reinterpret the memory to another type, without copying data. For example, we can view an `int64` array as an `int32` array with twice as many elements.

```
a = np.arange(6, dtype=np.int64)
array([0, 1, 2, 3, 4, 5])
a.view('i4') # 'i4' is short for integer with 4 bytes or int32
array([0, 0, 1, 0, 2, 0, 3, 0, 4, 0, 5, 0], dtype=int32)
# note this will not convert types,
# float will be interpreted differently
```

NumPy

Fancy indexing

Slicing is a great way to pull data from an array when the indices follow a regularly gridded pattern. But what if you want to pull out arbitrary indices? Or you wish to pull out indices that follow another pattern that is not regular, like the Fibonacci sequence? NumPy arrays handle these cases via **fancy indexing**.

```
a = 2*np.arange(8)**2 + 1
array([ 1,  3,  9, 19, 33, 51, 73, 99])

# pull out the fourth, last and second indices
a[[3, -1, 1]]
array([19, 99,  3])

# pull out the Fibonacci sequence
fib = np.array([0, 1, 1, 2, 3, 5])
a[fib]
array([ 1,  3,  3,  9, 19, 51])

# pull out an arbitrary 2x2 matrix
a[[[2, 7], [4, 2]]]
array([[ 9, 99],
       [33,  9]])
```

NumPy

Fancy indexing

You can mix slicing and fancy indexing.

```
a = np.arange(16) - 8
a.shape = (4, 4)
array([[-8, -7, -6, -5],
       [-4, -3, -2, -1],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7]])

# pull out the third, last and first columns
a[:, [2, -1, 0]]
array([[-6, -5, -8],
       [-2, -1, -4],
       [ 2,  3,  0],
       [ 6,  7,  4]])

# get the diagonal with a range
i = np.arange(4)
a[i, i]
array([-8, -3,  2,  7])
```

NumPy

Arithmetic

A defining feature of all array data languages is the ability to perform arithmetic operations in an *element-wise* way. This allows for concise mathematical expressions to be evaluated over an arbitrarily large amount of data. This works for scalars and for arrays with the same shape.

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

a - 1
array([-1,  0,  1,  2,  3,  4])

a + a
array([ 0,  2,  4,  6,  8, 10])

2*a**2 + 3*a + 1
array([ 1,  6, 15, 28, 45, 66])
```

NumPy

Arithmetic

Even though the above examples are extremely expressive, it can be subtly expensive. For each operation a new array is created and all elements are looped over. For the simple expression `a - 1` this overhead is fine. However, for the complex operation `2*a**2 + 3*a + 1` the allocation of new arrays is wasteful, since the are discarded immediately after the next operation is completed. Why create a special array for `a**2` if it is going to be deleted when you finish computing `2*a**2`?
These arrays are called **temporaries**.
Each operation iterates through all elements of the array on its own. So as an examples `6*a` would run about twice as fast as `3*(2*a)`. To remedy this problem there is the `numexpr`-module!

NumPy

Arithmetic

NumPy is still an incredibly expressive and powerful tool. Suppose you have to arrays `x` and `y` of the same shape. The numerical derivative `dy/dx` is given by this simple expression:

```
(y[1:] - y[:-1]) / (x[1:] - x[:-1])
```

NumPy

Arithmetic

NumPy is still an incredibly expressive and powerful tool. Suppose you have to arrays `x` and `y` of the same shape. The numerical derivative `dy/dx` is given by this simple expression:

```
(y[1:] - y[:-1]) / (x[1:] - x[:-1])
```

This method treats the points in `x` and `y` as bin boundaries and returns the derivative for the center points `((x[1:] + [x:-1])/2)`. As a side effect, however, the result is shorter by 1 that the length of the original arrays.

NumPy

Arithmetic

NumPy is still an incredibly expressive and powerful tool.
Suppose you have to arrays `x` and `y` of the same shape. The numerical derivative `dy/dx` is given by this simple expression:

```
(y[1:] - y[:-1]) / (x[1:] - x[:-1])
```

This method treats the points in `x` and `y` as bin boundaries and returns the derivative for the center points `((x[1:] + [x:-1])/2)`. As a side effect, however, the result is shorter by 1 that the length of the original arrays.
To treat the upper and lower bounds properly NumPy provides the `gradient()` function. The numerical derivative is then just:

```
np.gradient(y) / np.gradient(x)
```

NumPy

Broadcasting

Performing element-wise operations on arrays is not limited to scalars and arrays of the same shape. NumPy is able to *broadcast* arrays of different shapes together as long as their shapes follow some simple compatibility rules.

- the dimensions are equal, `a.shape[i] == b.shape[i]`
- the dimension is one, `a.shape[i] == 1` or `b.shape[i] == 1`
- the rank (number of dimensions) is less that that of the other array, `a.nbim < i or b.ndim < i`.

NumPy

Broadcasting

When the ranks of two axes of two arrays are equal, the operation between them is computed element-wise. This was the `a + a` example.

When the length of an axis is 1 on the array `a` and the length of the same axis on the array `b` is greater than 1, the value of `a` is virtually stretched along the entire length of `b`. This is where the term broadcasting comes from, since one element from `a` goes to all elements of `b`. Let us consider a 2x2 matrix times a 2x1 vector which broadcast the multiplication.

```
a = np.arange(4)
a.shape = (2, 2)
array([[0, 1],
       [2, 3]])
b = np.array([[42], [43]])
a * b
array([[  0,  42],
       [ 86, 129]])
```

Every column of `a` is multiplied element-wise by the values in `b`.

NumPy

Broadcasting

The above operation did not calculate the dot product. This requires the `dot()` function.

```
a = np.arange(4)
a.shape = (2, 2)
array([[0, 1],
       [2, 3]])
b = np.array([[42], [43]])
np.dot(a, b)
array([[ 43],
       [213]])
```

NumPy

Universal functions

Now we know how to define and manually manipulate arrays. NumPy has what are called *universal functions* that work on arrays.

```
add(a, b) # addition operator +
subtract(a, b) # Subtraction operator -
multiply(a, b) # Multiplication operator *
divide(a, b) # Division operator /
power(a, b) # Power operator **
mod(a, b) # Modulus
abs(a) # Absolute value
sqrt(a) # Positive square root
conj(a) # Complex conjugate
exp(a) # Exponential e**a
exp2(a) # Exponential with base 2
log(a) # Natural log
log2(a) # Log base 2
log10(a) # Log base 10
sin(a) # Sine
cos(a) # Cosine
tan(a) # tangent

...
```

NumPy

Universal functions continued

```
...

bitwise_or(a, b) # Bitwise | operator
bitwise_xor(a, b) # Bitwise ^ operator
bitwise_and(a, b) # Bitwise & operator
invert(a) # Bitwise inversion, i.e. the ~ operator
left_shift(a, b) # Left bit shift operator <<
right_shift(a, b) # Right bit shift operator >>
minimum(a, b) # Minimum, this is not np.min()
maximum(a, b) # Maximum, this is not np.max()
# minimum and maximum create a new array from both input arrays
# min and max return the min or max value from one array
isreal(a) # Test for zero imaginary component
iscomplex(a) # Test for zero real component
isfinite(a) # Test for non-infinite value
isinf(a) # Test for infinite value
isnan(a) # Test for Not-A-Number
floor(a) # Next lowest integer
ceil(a) # Next highest integer
trunc(a) # Truncate, remove non-integer bits
```

NumPy

Universal functions

Do not confuse the NumPy functions with their `math` counterparts.
NumPy's functions are designed to work with arrays, while the `math` function focus on floats.
The NumPy versions are usually faster.

```
x = np.linspace(0.0, np.pi, 5)
array([ 0., 0.78539816, 1.57079633, 2.35619449, 3.14159265])
np.sin(x)
array([ 0.00000000e+00, 7.07106781e-01,
        1.00000000e+00, 7.07106781e-01, 1.22464680e-16])
```

NumPy

Other functions

In addition to the universal functions, NumPy provides a number of useful functions for day-to-day use. Many of these allow you to supply keyword arguments, such as an `axis` to work on. It defaults to `None` meaning the function will operate over the entire array. However, if `axis` is an integer or a tuple of integers the function will only operate over those dimensions.

```
a = np.arange(9)
a.shape = (3, 3)
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

np.sum(a)
36

np.sum(a, axis=0)
array([ 9, 12, 15])

np.sum(a, axis=1)
array([ 3, 12, 21])
```

NumPy

Useful functions

Here is a list of some more useful NumPy functions.

```
sum(a) # adds all elements
prod(a) # multiplies all elements
min(a) # returns the smallest element
max(a) # returns the largest element
argmin(a) # return the location (index) of the minimum element
argmax(a) # returns the location (index) of the maximum element
dot(a, b) # dot product of two arrays
cross(a, b) # cross product of two arrays
mean(a) # mean value of the array elements
median(a) # median value of the array elements
average(a, weights=None) # Weighted average
std(a) # standard deviation
var(a) # variance

...
```

NumPy

Useful functions continued

```
...

unique(a) # Sorted unique elements of an array
asarray(a, dtype) # Ensure the array is of a given type.
                  # If the array is already in the specified type,
                  # no copy is made.
atleast_1d # Ensures that the array is at least one dimensional
atleast_2d # Ensures that the array is at least two dimensional
atleast_3d # Ensures that the array is at least three dimensional
append(a, b) # Create new array by combining two arrays
save(file, a) # Save array to disk (binary format) .npy
load(file) # Read array from disk (binary format) .npy
memmap(file) # Load an array from disk lazily
```

NumPy

Examples

A classical linear algebra example is solving a system of equations.

$$\begin{aligned} 3x + 6y - 5z &= 12 \\ x - 3y - 2z &= -2 \\ 5x - y + 4z &= 10 \end{aligned}$$

We can express this in matrix form as:

$$\begin{bmatrix} 3 & 6 & -5 \\ 1 & -3 & 2 \\ 5 & -1 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 12 \\ -2 \\ 10 \end{bmatrix}$$

Now let us solve this using two different ways with NumPy.
We have to solve $\mathbf{AX} = \mathbf{B}$ for the variable $\mathbf{X}$.
This means we need to obtain $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$.

NumPy

Examples

A straight-forward solution involves using the `matrix` class of NumPy.

```
import numpy as np

# Defining the matrices
A = np.matrix([[3, 6, 5],
               [1, -3, 2],
               [5, -1, 4]])

B = np.matrix([[12],
               [-2],
               [10]])

# solving for the variable by inverting A
```

NumPy

Examples

A straight-forward solution involves using the `matrix` class of NumPy.

```
import numpy as np

# Defining the matrices
A = np.matrix([[3, 6, 5],
               [1, -3, 2],
               [5, -1, 4]])

B = np.matrix([[12],
               [-2],
               [10]])

# solving for the variable by inverting A

X = A ** (-1) * B
print(X)

# matrix([[1.75],
#         [1.75],
#         [0.75]])
```

NumPy

Examples

The Pythonic solution is utilizing NumPy's prime data type: The array!

```
import numpy as np

# Defining the arrays
a = np.array([[3, 6, -5],
              [1, -3, 2],
              [5, -1, 4]])

b = np.array([12, -2, 10])

# Solving for the variable using linalg functions
```

NumPy

Examples

The Pythonic solution is utilizing NumPy's prime data type: The array!

```
import numpy as np

# Defining the arrays
a = np.array([[3, 6, -5],
              [1, -3, 2],
              [5, -1, 4]])

b = np.array([12, -2, 10])

# Solving for the variable using linalg functions

x = np.linalg.inv(a).dot(b)
print(x)

# array([1.75, 1.75, 0.75])
```

NumPy

Examples

Both versions are valid solutions. But which one is the best?
The `numpy.matrix` method is syntactically the simplest. However, `numpy.array` is the most practical. Since NumPy primarily revolves around arrays, converting matrices to arrays back and forth can become cumbersome. Using only one data type in your program reduces headaches and bugs. In the above example the `array` version is also faster.
My advice is to always use arrays when working with NumPy.

NumPy

Examples

Here are more examples from linear algebra using arrays.

```
# calculating the determinate of a 'matrix'
a = np.array([[4, 2, 0],
              [9, 3, 7],
              [1, 2, 1]])
np.linalg.det(a)
# result: -48.000000000000028


# Calculating eigenvalues and eigenvectors

vals, vecs = np.linalg.eig(a)

vals
array([ 8.85591316,  1.9391628 , -2.79507597])

vecs
array([[-0.3663565 , -0.54736745,  0.25928158],
       [-0.88949768,  0.5640176 , -0.88091903],
       [-0.27308752,  0.61828231,  0.39592263]])
```

NumPy

Examples

NumPy also supplies methods for working with polynomials. Given a set of roots, it is possible to calculate the polynomial coefficients.

```
>>> np.poly([-1, 1, 1, 10])
array([  1, -11,   9,  11, -10])
```

The returned array corresponds to the following polynomial.
$x^4 - 11x^3 + 9x^2 + 11x - 10$.

NumPy

Examples

NumPy also supplies methods for working with polynomials. Given a set of roots, it is possible to calculate the polynomial coefficients.

```
>>> np.poly([-1, 1, 1, 10])
array([  1, -11,   9,  11, -10])
```

The returned array corresponds to the following polynomial.
$x^4 - 11x^3 + 9x^2 + 11x - 10$.

The opposite operation can be performed as well. Given a set of coefficients, the `root` function returns all of the polynomial roots:

```
>>> np.roots([1, 4, -2, 3])
array([-4.57974010+0.j,  0.28987005+0.75566815j,
        0.28987005-0.75566815j])
```

Notice here that the roots of the polynomial $x^3 + 4x^2 - 2x + 3$ are imaginary.

NumPy

Examples

We can use coefficient arrays to integrate polynomials.
Consider integrating $x^3 + x^2 + x + 1$ to
$x^4/4 + x^3/3 + x^2/1 + x + C$. By default, the constant C is set to zero:

```
>>> np.polyint([1, 1, 1, 1])
array([0.25, 0.33333333, 0.5, 1., 0.])
```

NumPy

Examples

We can use coefficient arrays to integrate polynomials. Consider integrating $x^3 + x^2 + x + 1$ to $x^4/4 + x^3/3 + x^2/1 + x + C$. By default, the constant C is set to zero:

```
>>> np.polyint([1, 1, 1, 1])
array([0.25, 0.33333333, 0.5, 1., 0.])
```

Similarly, derivatives can be taken:

```
>>> np.polyder([1./4., 1./3., 1./2., 1., 0.])
array([ 1.,  1.,  1.,  1.])
```

The functions `polyadd`, `polysub`, `polymul`, and `polydiv` also handle proper addition, subtraction, multiplication, and division of polynomial coefficients, respectively.

NumPy

Examples

To evaluate a polynomial at a particular point, we can use the `polyval` method. Consider $x^3 - 2x^2 + 2$ evaluated at $x = 4$.

```
>>> np.polyval([1, -2, 0, 2], 4)
34
```

NumPy

Examples

To evaluate a polynomial at a particular point, we can use the `polyval` method. Consider $x^3 - 2x^2 + 2$ evaluated at $x = 4$.

```
>>> np.polyval([1, -2, 0, 2], 4)
34
```

Finally, the `polyfit` function can be used to fit a polynomial of specified order to a set of data using a least-squares approach:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> y = [0, 2, 1, 3, 7, 10, 11, 19]
>>> np.polyfit(x, y, 2)
array([ 0.375, -0.88690476, 1.05357143])
```