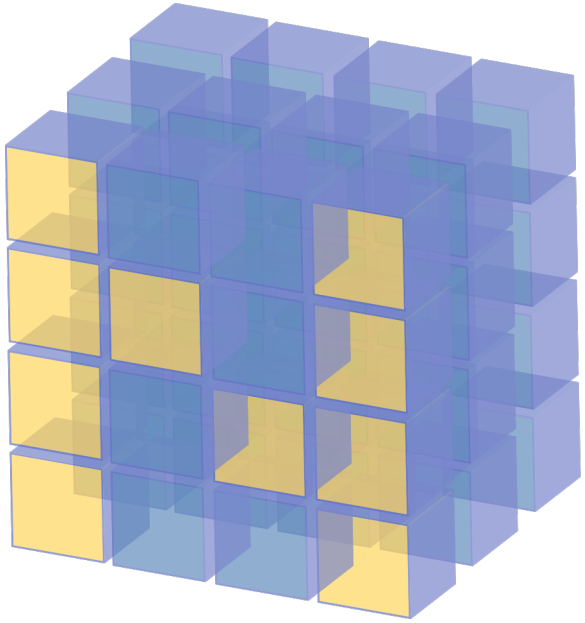
Goals for Today:

Numpy:

- Arrays
- Initializing
- Operators
- Indexing
- Numpy functions



NumPy

Numpy Overview

Numpy is a ubiquitous module in scientific python applications that supplies an important class called an **array**.

Arrays are essentially numerical implementations of tensors (i.e., vectors, matrices, and other higher dimensional numerical objects).

We will assume throughout these examples that we have imported numpy:

```
import numpy as np
```

Thus, whenever we call numpy objects we will use the namespace `np.object`

Historical Note: Scipy is a very common module that also includes numpy. This is because numpy preceeded scipy, and scipy builds on it. If you are looking up things online you will often see them used interchangably, but the basic distinction is that Scipy contains a lot of additional functions and numerical methods that are not part of numpy.

Initializing Arrays – with Lists

Initializing a numpy array is as simple passing a list to the `np.array()` constructor (class `__init__`, looks like a function):

```
a = np.array(range(9))
print(a)
```

```
[0 1 2 3 4 5 6 7 8]
```

The array function accepts a list containing a single datatype, and has an optional argument for the datatype `dtype`:

```
b = np.array(range(9),dtype=np.float32)
print(b)
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8.]
```

When we supply the type `np.float32`, the list of integers returned by `range(5)` are upcast to numpy 32-bit floats (*Note: built-in python floats are 64-bit, taking up more space but providing more precision*).

Note: built-in python floats are 64-bit, taking up more space but providing more precision).

Initializing Arrays – with Lists

Arrays only hold data of a *single type*, if you try to supply mixed numerical types it will always **up-cast** to the highest level required to represent all of the objects:

```
c = np.array([1,2.0,1+1j])
d = np.array([1,2.0,"3"])
print(c)
print(d)
```

```
[1.+0.j 2.+0.j 1.+1.j]
['1' '2.0' '3']
```

In the first example all of the numbers are recast as complex, in the second they are recast as strings.

Arrays can also be initialized with tupels:

```
d = np.array((1,2,3))
print(d)
```

```
[1 2 3]
```

Note: You won't get an error if you pass `sets` and `dictionaries` to `np.array()`, but their behavior won't be array-like.

Initializing Arrays – with Lists

You can also initialize arrays to have multiple dimensions, by initializing them with lists of lists:

```
e = np.array([1,2]) # 1D array (vector)
f = np.array([[1,2],[3,4]]) # 2D array (matrix)
g = np.array([[[1,2],[3,4]],[[5,6],[7,8]]]) # 3D array(tensor)
print("1D:\n{}".format(e))
print("\n2D:\n{}".format(f))
print("\n3D:\n{}".format(g))
```

```
1D:
[1 2]

2D:
[[1 2]
 [3 4]]

3D:
[[[1 2]
 [3 4]]

 [[5 6]
 [7 8]]]
```

Initializing Arrays – with Functions

Numpy also comes with several built-in functions for initializing arrays:

- `zeros([n_rows,n_cols,...],dtype=float64)` can be used to initialize an array of zeroes with the specified shape and type.

- `ones([n_rows,n_cols,...],dtype=float64)` can be used to initialize an array of ones with the specified shape and type.

- `full([n_rows,n_cols,...],value)` can be used to initialize an array of values with the specified shape and type.

- `arange(start=0,stop,step=1,dtype=int64)` can be used to initialize a vector with evenly spaced between `start` and `stop` with a spacing of `step`, excluding the `stop` value but including the `start` value. `start` and `step` are optional.

- `linspace(start=0,stop,num=50)` can be used to initialize a vector with a specified number of values between the `start` and `stop` values. Similar to `arange` but you specify the number of values rather than the step.

Initializing Arrays – with Functions

The behavior of these functions is more clearly illustrated with examples:

```
h = np.zeros([2,2])
i = np.ones([2,2])
j = np.full([2,2],100.)
k = np.arange(10)
l = np.linspace(1,2,num=11)
print(h)
print(i)
print(j)
print(k)
print(l)
```

```
[[[0. 0.]
  [0. 0.]]
[[1. 1.]
  [1. 1.]]
[[100. 100.]
  [100. 100.]]
[0 1 2 3 4 5 6 7 8 9]
[1. 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2. ]
```

Numpy has even more functions for initializing arrays for particular purposes, but these are the most common.

Initializing Arrays – from Files

Numpy also supplies a relatively flexible built-in function called `loadtxt(filename,comments='#',delimiter=' ',skiprows=0,...)` for initializing arrays from suitably delimited files:

```
m = np.loadtxt('sample.txt',comments="!",skiprows=1)
print(m)
```

```
[[ 0. 12.71 ]
 [ 1. 14.224]
 [ 2. 20.238]
 [ 3. 18.982]]
```

The contents of `sample.txt`:

```
val1 val2
0.000 12.710
1.000 14.224
2.000 20.238
3.000 18.982 ! this is a comment
```

Here the `skiprows` argument was used to avoid the header, and the `comments` argument was used to parse the last row without error.

Initializing Arrays – from Files

`genfromtxt()` is another numpy function with similar arguments that can also apply simple rules for missing values:

```
n = np.genfromtxt('sample2.txt',comments="!",skip_header=1,\
                  filling_values=100)
print(n)
```

```
[[ 0.  12.71 ]
 [ 1.  14.224]
 [ 2. 100.   ]
 [ 3.  18.982]]
```

The contents of `sample2.txt`:

```
 val1  val2
0.000 12.710
1.000 14.224
2.000 _____
3.000 18.982     ! this is a comment
```

Here the non-numeric missing entry (`_____`) was replaced with the specified value.

Array Indexing

Arrays are 0-indexed like every other data structure in Python.

You can access values from arrays using the same slicing notation that we use for lists:

```
m = np.arange(10)
print(m[2::2])
```

```
[2 4 6 8]
```

For multi-dimensional arrays, you can use sequential `[]` notation to access specific elements and rows. The first index corresponds to rows, the second to columns, etc. The result of each `[]` is returned *before* the next is evaluated.

```
n = np.array([[1,2],[3,4]])
print(n)
print(n[1][1])
print(n[0])
```

```
[[1 2]
 [3 4]]
4
[1 2]
```

Array Indexing

If you want to return specific columns or other dimensions you would use `[row,col,...]` notation.

This is distinct from the sequential notation in that the slicing operation is carried out in a single step:

```
o = np.array([[1,2],[3,4]])
print(o)
print(o[:,1]) # Example 1
print(o[:][1]) # Example 2
```

```
[[1 2]
 [3 4]]
[2 4]
[3 4]
```

In the first example the second column is returned, but in the second example the second row is returned. The latter result occurs because the first slicing operation `o[:]` just returns the full array, with the result that the second slicing operation (`[1]`) returns the second row.

Array Indexing

Finally, arrays are distinct from lists in that you can pass a list of indices to arrays via the `[]` notation to return values in that order:

```
o = np.array([[1,2],[3,4]])
print(o[[1,0]])
print(o[[1,0,1],[1,0,0]])
```

```
[[3 4]
 [1 2]]
[4 1 3]
```

In the first case, the rows are returned in order specified in the list.

In the second case where two lists are passed, the first list specifies the row locations and the second list specifies the column locations (i.e., the `o[1][1]`, `o[0][0]`, and `o[1][0]` elements).

The second case is a useful reference for later when we discuss using the `where()` function.

Array Operator Behavior

Numpy arrays behave within typical operators (`+,-,*,**,/,%,>,==`) in **element-wise** fashion.

Thus, `a*b` **does not** mean dot product, it means **element-wise** product between the arrays pointed to by `a` and `b`:

```
p = np.arange(4)
print("p: {}".format(p))
print("p+p: {}".format(p+p))
print("p-p: {}".format(p-p))
print("p*p: {}".format(p*p))
print("p**p: {}".format(p**p))
print("p/p: {}".format(p/p))
print("p%p: {}".format(p%p))
print("p>p: {}".format(p>p))
print("p==p: {}".format(p==p))
```

```
p: [0 1 2 3]
p+p: [0 2 4 6]
p-p: [0 0 0 0]
p*p: [0 1 4 9]
p**p: [ 1 1 4 27]
p/p: [nan 1. 1. 1.]
p%p: [0 0 0 0]
p>p: [False False False False]
p==p: [ True True True True]
```

Note: when you run this example you will get a warning about division by zero (that is responsible for the `nan` value in `p/p`).

Since operators act **element-wise**, they only make sense when `a` and `b` point to arrays of the same shape (see **broadcasting** behavior in numpy docs for more details on when operations can be performed on arrays of different size).

Array Operator Behavior

You can also apply operators between arrays and scalars, in which case the same scalar is applied to every element of the array in the operation:

```
print("p: {}".format(p))
print("p+1.0: {}".format(p+1.0))
print("p-1.0: {}".format(p-1.0))
print("p*2.0: {}".format(p*2.0))
print("p**2.0: {}".format(p**2.0))
print("p/2.0: {}".format(p/2.0))
print("p%2.0: {}".format(p%2.0))
print("p>2: {}".format(p>2))
print("p==2: {}".format(p==2))
```

```
p: [0 1 2 3]
p+1.0: [1. 2. 3. 4.]
p-1.0: [-1. 0. 1. 2.]
p*2.0: [0. 2. 4. 6.]
p**2.0: [0. 1. 4. 9.]
p/2.0: [0. 0.5 1. 1.5]
p%2.0: [0. 1. 0. 1.]
p>2: [False False False True]
p==2: [False False True False]
```

Note: operators always return an array of the same shape. In the case of logical operators, the result is an array of booleans. Upcasting rules also apply.

Numpy also introduces a new operator, `@`, for the dot product:

```
print("p@p: {}".format(p@p))
```

```
14
```

This is equivalent to `array.dot()` and `dot()` introduced later. Arrays must be compatible and the shape of the returned array depends on inputs.

Array Methods - Reshaping

Numpy arrays come with a number of built-in methods for **reshaping** arrays, **linear-algebra** operations, calculating **statistics**, **sorting**, and performing **logical** comparisons.

Reshaping: When we initialize an array, we can reorganize the elements using the `.reshape(dim1,dim2,...)` method to convert a vector into a matrix, etc. and vice versa:

```
# Example 1: Using reshape to increase dimensions
q = np.arange(9).reshape(3,3)
print(q)

# Example 2: Using reshape to reduce dimensions
print(q.reshape(9))
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[0 1 2 3 4 5 6 7 8]
```

Note: when increasing dimensions, the default behavior fills the last dimension first (i.e., columns in example 1). If you want to fill rows first you can use the `order='F'` argument.

Array Methods – Linear Algebra

One of the major motivations for developing numpy was to provide python users with access to objects that behave like tensors.

The numpy arrays thus supply methods for dot product, transpose, and complex conjugate:

```
r = np.linspace(1+1J,4+4J,num=4).reshape(2,2)
print(r)
print(r.dot(r))
print(r.dot(r.transpose()))
print(r.dot(r.transpose().conj())) # Hermitian
```

```
[[1.+1.j 2.+2.j]
 [3.+3.j 4.+4.j]]
[[0.+14.j 0.+20.j]
 [0.+30.j 0.+44.j]]
[[0.+10.j 0.+22.j]
 [0.+22.j 0.+50.j]]
[[10.+0.j 22.+0.j]
 [22.+0.j 50.+0.j]]
```

Note: `a.dot()` is equivalent to a•b (not b•a).

More linear algebra operations can be applied using numpy functions.

Array Methods – Statistics

Arrays come with built-in methods for calculating means, sums, products, and standard deviations along rows/columns:

```
# Mean Examples
r = np.arange(4).reshape(2,2)
print("r:\n{}".format(r))
print("\nmean examples:")
print(r.mean())
print(r.mean(axis=0))
print(r.mean(axis=1))

# Sum Examples
print("\nsum examples:")
print(r.sum())
print(r.sum(axis=0)) # over rows
print(r.sum(axis=1)) # over columns

# Product Examples
print("\nprod examples:")
print(r.prod())
print(r.prod(axis=0)) # over rows
print(r.prod(axis=1)) # over columns

# Stdev Examples
print("\nstd examples:")
print(r.std())
print(r.std(axis=0)) # over rows
print(r.std(axis=1)) # over columns
```

```
r:
[[0 1]
 [2 3]]

mean examples:
1.5
[1. 2.]
[0.5 2.5]

sum examples:
6
[2 4]
[1 5]

prod examples:
0
[0 3]
[0 6]

std examples:
1.118033988749895
[1. 1.]
[0.5 0.5]
```

These methods accept an optional `axis` argument that indicates which dimension you want to perform the operation over. Default is all elements.

Array Methods – Sorting

Arrays come with built-in methods for sorting values and returning the indices of the sorted array:

```
# Initialize 2D array
s = np.array([[4,1],[2,3]])
print("s:\n{}\n".format(s))

# Example 1: Sort rows
s = np.array([[4,1],[2,3]])
s.sort() # Same as axis=1
print("\ns.sort():\n{}".format(s))

# Example 2: Sort columns
s = np.array([[4,1],[2,3]])
s.sort(axis=0)
print("\ns.sort(axis=0):\n{}".format(s))
```

```
s:
[[4 1]
[2 3]]

s.sort():
[[1 4]
 [2 3]]

s.sort(axis=0):
[[2 1]
 [4 3]]
```

Example 1: The `.sort()` method modifies the existing array and does not return a new array. By default `axis=-1` (i.e., the last dimension of the array). In this case, this is equivalent to `axis=1`, so the sorting is across the columns and each row has the same values but in ascending order.

Example 2: `axis=0`, so the sorting is across the rows and each column has the same values but in ascending order.

Array Methods – Sorting

Arrays also come with the `.argsort()` method that returns the indices that would sort the array.

This can be useful when you want to sort an array by the values in a particular column or row:

```
t = np.array([[3,0.1],[1,0.2],[2,0.2]])
inds = t[:,0].argsort()
print("t:\n{}\n".format(t))
print("inds:\n{}\n".format(inds))
print("t[inds]:\n{}".format(t[inds]))
```

```
t:
[[3.  0.1]
 [1.  0.2]
 [2.  0.2]]

inds:
[1 2 0]

t[inds]:
[[1.  0.2]
 [2.  0.2]
 [3.  0.1]]
```

In this example, suppose that the first column in the array were time and we wanted to sort the rest of values in each row so that the times are ordered.

Here, `t[:,0].argort()` returns a list of indices (`inds`) that sorts the times. The array can then be called with these indices (`t[inds]`) to return the array sorted by the time values.

Array Methods – Logic

When you want to use logical operators on arrays like `>` and `==` you might want to know if "any" value in the array is "greater than" or "equal to" a value, respectively. Alternatively, you might want to know if "all" values in the array satisfy the comparison.

The `.all()` and `.any()` methods are used for this purpose:

```
u = np.array([3,3,4,5])
print((u>3).any())
print((u>3).all())
```

```
True
False
```

Example 1: `(u>3)` returns an array of booleans (recall **array operator** behavior) which is passed to `.any()` for evaluation. `.any()` returns `True` because `4` and `5` are greater than `3`.

Example 2: `(u>3)` returns an array of booleans which is passed to `.all()` for evaluation. `.all()` returns `False` because the first two elements are `False` (i.e., not "all" of the booleans are `True`).

Array Functions – `where()`

Even though Numpy is the "light-weight" version of "scipy", it comes packed with a lot of useful functions for arrays relevant to finding specific elements and performing additional linear algebra operations.

The `np.where(condition)` function is a flexible and optimized function for finding the indices in an array where a condition is `True`:

```
v = np.array([[3,1],[2,20]])
print(np.where(v>1))
print(v[np.where(v>1)])
```

```
(array([0, 1, 1]), array([0, 0, 1]))
[ 3  2 20]
```

`np.where(v>1)` returns a tuple of two arrays with the row and column positions corresponding to where `v>1`. The tuple contains two arrays because `v` is two dimensional. For example, on a vector `np.where` only returns one array:

```
w = np.array([1,2,3,4])
print(np.where(w == 1))
print(w[np.where(w == 1)])
```

```
(array([0]),)
[1]
```

Array Functions – Linear Algebra

Some array methods can be used to perform linear algebra operations, but a more general set is available through the numpy methods. cross(), dot(), outer() all correspond to the expected matrix operations:

```
# cross, dot, and outer product examples
x = np.array([1,0,0])
y = np.array([0,1,0])
print(np.cross(x,y))
print(np.dot(x,y))
print(np.outer(x,y)) # analogous to dot(x.T,y)
```

```
[0 0 1]
0
[[0 1 0]
 [0 0 0]
 [0 0 0]]
```