Problem-set: NumPy at Trento Autumn School 2010

These files may be downloaded from **https://portal.g-node.org/python-autumnschool/materials/advanced_numpy**

Please do explore beyond the problems given, and feel free to ask questions at any time.

Note

Solutions to some problems are provided, but try to avoid peeking at them until you've made an attempt!

Before we start, load NumPy:

```
import numpy as np
```

The NumPy array

These exercises explore some of the more advanced features of NumPy. If you prefer to learn the basics from scratch, have a look at **this introduction**.

1. Let's explore the basics of a NumPy array. Create a 3x4 array of random values (using `np.random.random`), and call it `x`. Create another array as follows: `y = x[2]`. What happens when you modify `y`–does `x` also change?

2. In the array above, have a look at its different properties: `x.shape`, `x.ndim`, `x.dtype`, `x.strides`. What does each property tell you about the array?

3. Compare the `strides` property of `x.T` to the above. What is `x.T` and can you explain the new strides?

4. Compute the sum of the columns of `x`. You may find the function to do that by using `np.<TAB>` in IPython or `print dir(np)` in standard Python. Remember that you can always look at the docstring of any function in Python (use `np.func_name?<ENTER>` in IPython or `help(np.func_name)` in standard Python).

5. Construct the array `x = np.array([0, 1, 2, 3], dtype=np.uint8)` (here, `uint8` represents a single byte in memory, an unsigned integer between 0 and 255). Can you explain the results obtained by `x + 1` and `x / 2`? Also try `x.astype(float) + 1` and `x.astype(float) / 2`.

6. What is the maximum number of dimensions a NumPy array can have? Use one of the array constructors (`np.zeros`, `np.empty`, `np.random.random`, etc.) to find out.

Structured Arrays

1. Design a data-type for storing the following record:

   - Timestamp in nanoseconds (a 64-bit unsigned integer)
   - Position (x- and y-coordinates, stored as floating point numbers)

   Use it to represent the following data:

   ```
   dt = np.dtype(<your parameters here>)
   x = np.array([(100, (0, 0.5)),
                 (200, (0, 10.3)),
                 (300, (5.5, 15.1))], dtype=dt)
   ```

   Have a look at the `np.dtype` docstring if you need help. After constructing `x`, try to print all the x values for which time is greater than 100 (hint: something of the form `y[y > 100]`).

2. Consider structured_arrays/data.txt. Modify structured_arrays/load_txt_template.py to load the data in this text file. This requires specifying a data-type that encapsulates a record such as

   ```
   # name  x      y      block - 2x3 ints
   aaaa    1.0    8.0    1 2 3 4 5 6
   ```

Note that binary files can also be read with structured data-types, using np.fromfile.

3. Create two structured arrays of your choosing. Use the np.savez command to save these to a single data-file. Load the data-file using np.load and confirm whether the data survived the round-trip. (Saving data using save or savez is highly recommended over pickling.)

Broadcasting

1. Take a look at the following. Many of you may know it as a meshgrid:

   ```
   x, y = np.mgrid[:10, :5]
   z = x + y
   ```

   Now, reproduce z from the above snippet, but this time use NumPy's ogrid together with broadcasting.

2. In our solution, broadcasting is used "behind the scenes". To see what happens more explicitly, apply np.broadcast_arrays on the x and y given by ogrid ogrid. This should correspond to the x and y produced by mgrid.

3. Benchmark the two approaches (one using mgrid, the other ogrid), using IPython's timeit function. Can you explain the difference in execution time?

4. Given a list of 3-dimensional coordinates, p,

   ```
   [[1.0, 2.0, 10],
    [3.0, 4.0, 20],
    [5.0, 6.0, 30],
    [7.0, 8.0, 40]]
   ```

   Normalise each coordinate by dividing with its Z (3rd) element. For example, the first row becomes:

   ```
   [1/10, 2/10, 10/10]
   ```

   Hint: extract the last column as y, and then change its dimensions so that p / y works.

Indexing

1. Create a 3x3 ndarray. Use fancy indexing to slice out the diagonal elements.

2. Predict and verify the shape of the following slicing operation.

   ```
   x = np.empty((10, 8, 6))

   idx0 = np.zeros((3, 8)).astype(int)
   idx1 = np.zeros((3, 1)).astype(int)
   idx3 = np.zeros((1, 1)).astype(int)

   x[idx0, idx1, idx2]
   ```

3. **Advanced**: This is not strictly speaking a question on indexing, but it's a fun exercise either way.

   Construct an array

   ```
   x = np.arange(12, dtype=np.int32).reshape((3, 4))
   ```

   so that x is

   ```
   array([[ 0,  1,  2,  3],
          [ 4,  5,  6,  7],
          [ 8,  9, 10, 11]])
   ```

   Now, provide to np.lib.stride_tricks.as_strided the strides necessary to view a sliding 2x2 window over this array. The output should be

   ```
   array([[[[ 0,  1],
            [ 4,  5]],
   ```

```
       [[ 1,  2],
        [ 5,  6]],

       [[ 2,  3],
        [ 6,  7]]],


      [[[ 4,  5],
        [ 8,  9]],

       [[ 5,  6],
        [ 9, 10]],

       [[ 6,  7],
        [10, 11]]]], dtype=int32)
```

The code is of the form

```
z = as_strided(x, shape=(2, 3, 2, 2),
                  strides=(..., ..., ..., ...))
```

This sort of stride manipulation is very handy to implement techniques such as region based statistics, convolutions, etc.

Array Interface

Documentation for NumPy's `__array_interface__` may be found in the **online docs**.

1. An author of a foreign package (array_interface/mutable_str.py) provides a string class that allocates its own memory:

```
In [1]: from mutable_str import MutableString

In [2]: s = MutableString('abcde')

In [3]: print s
abcde
```

You'd like to view these mutable strings as ndarrays, in order to manipulate the underlying memory.

   a. Add an `__array_interface__` dictionary attribute to `s`, then convert `s` to an ndarray. Use the given array_interface/template.py as a guide.
   b. Add "1" to the array. Now print the original string to ensure that its value was modified.

Advanced Exercises

1. Construct the following two arrays:

```
x = np.array([[1, 2], [3, 4]], order='C', dtype=np.uint8)
y = np.array([[1, 2], [3, 4]], order='F', dtype=np.uint8)
```

Compare the bytes they store in memory by using

```
[ord(c) for c in x.data]
```

Note that, even though these arrays store data in different memory order, they are identical from the user's perspective.

```
print x
print y
```

2. Examine the bytes stored by the following array (using the "ord" trick shown above).

```
x = np.array([[1, 2], [3, 4]], dtype=np.uint32)
```

Note that, on most laptops, the byte order will be little Endian, i.e. least significant byte first.

3. Attempt the **Fortran-ordering quiz**.