Matrix Programming

Basics

NumPy (http://numpy.scipy.org)
Provides MATLAB-style arrays for Python
And many other things
A data parallel programming model
  - Write $x^A x^T$ to calculate $x A x^T$
  - The computer takes care of the loops
All encapsulated in special objects called arrays
Create an array from a list

```python
>>> import numpy
>>> vals = [1, 2, 3]
>>> arr = numpy.array(vals)
>>> arr
array([1, 2, 3])
```

Arrays are **homogeneous**
- I.e., all values have the same type
- Allows values to be packed together
- Saves memory
- Faster to process
So what does this do?

```python
>>> arr = numpy.array([1, 2.3])
>>> arr
array([1., 2.3])
```

A float, not an int

You can specify at creation time:

```python
>>> array([1, 2, 3, 4], dtype=float32)
array([ 1., 2., 3., 4.])
```
You can specify at creation time:
```python
>>> array([1, 2, 3, 4], dtype=float32)
array([ 1., 2., 3., 4.])
```

Why would you want to specify a type?

You can also specify the data type later
```python
>>> a = array([1, 2, 3, 4], dtype=float32)
>>> a.astype(int)
array([ 1, 2, 3, 4])
>>> a.dtype = int
>>> a
array([1065353216, 1073741824, 1077936128, 1082130432])
```
Basic data types (in increasing order) are:

- `bool`
- `uint[8,16,32,64]`
- `int`
- `float`
- `int8`
- `float[32,64,128]`
- `int16`
- `complex`
- `int32`
- `complex[64,126]`
- `int64`

Many other ways to create arrays

```python
>>> z = numpy.zeros((2, 3))
>>> z
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Type is `float` unless something else specified.
Many other ways to create arrays

```python
>>> z = numpy.zeros((2, 3))
>>> z
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Type is float unless something else specified

What do these do?

```python
>>> block = numpy.ones((4, 5))
>>> mystery = numpy.identity(4)
```

Can create arrays without filling in values

```python
>>> x = numpy.empty((2, 2))
>>> x
array([[3.82265e-297, 4.94944e+173],
       [1.93390e-309, 1.00000e+000]])
```

"Values" will be whatever bits were in memory

Should not be used without being initialized
Can create arrays without filling in values

```python
>>> x = numpy.empty((2, 2))
>>> x
array([[3.82265e-297, 4.94944e+173],
       [1.93390e-309, 1.00000e+000]])
```

"Values" will be whatever bits were in memory

Should not be used without being initialized

When is this useful?

Assigning creates alias: does not copy data

```python
>>> first = numpy.ones((2, 2))
>>> first
array([[1., 1.],
       [1., 1.]])
```

```python
>>> second = first
>>> second[0, 0] = 9
>>> first
array([[9., 1.],
       [1., 1.]])
```
Assigning creates alias: does not copy data

```python
>>> first = numpy.ones((2, 2))
```

```python
array([[1., 1.],
       [1., 1.]])
```

```python
>>> second = first
```

```python
array([[1., 1.],
       [1., 1.]])
```

```python
>>> second[0, 0] = 9
```

```python
Not second[0][0]
```

```python
>>> first
```

```python
array([[9., 1.],
       [1., 1.]])
```

Use the `array.copy` method

```python
>>> first
```

```python
array([[1., 1.],
       [1., 1.]])
```

```python
>>> second = first.copy()
```

```python
>>> second[0, 0] = 9
```

```python
>>> first
```

```python
array([[1., 1.],
       [1., 1.]])
```
Arrays also have properties

```python
>>> first
array([[1., 1.],
       [1., 1.]])

>>> first.shape
(2, 2)

>>> block = numpy.zeros((4, 7, 3))

>>> block.shape
(4, 7, 3)
```
Arrays also have properties

```python
>>> first
array([[1., 1.],
       [1., 1.]])
>>> first.shape
(2, 2)
>>> block = numpy.zeros((4, 7, 3))
>>> block.shape
(4, 7, 3)
```

`array.size` is the total number of elements

```python
>>> first.size
4
>>> block.size
84
```
**Matrices Basics**

Reverse on all axes with `array.transpose`

```python
>>> first = numpy.array([[1, 2, 3],
                        [4, 5, 6]])
```

```python
>>> first.transpose()
array([[1, 4],
       [2, 5],
       [3, 6]])
```

```python
>>> first
array([[1, 2, 3],
       [4, 5, 6]])
```

**Flatten arrays using `array.ravel`**

```python
>>> first = numpy.zeros((2, 2, 2))
>>> second = first.ravel()
```

```python
>>> second.shape
(8,)
```
Think about the 2×4 array A:

```python
>>> A
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

A looks 2-dimensional
But computer memory is 1-dimensional
Must decide how to lay out values

*Row-major order* concatenates the rows
Used by C and Python

<table>
<thead>
<tr>
<th>Logical</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>5 6 7 8</td>
<td></td>
</tr>
</tbody>
</table>
Column-major order concatenates the columns
Used by Fortran and MATLAB

No difference in usability or performance...
...but causes headaches when passing data from one language to another
(Just like 0-based vs. 1-based indexing)
No difference in usability or performance...  
...but causes headaches when passing data from one language to another  
(Just like 0-based vs. 1-based indexing)

What order are 3-dimensional arrays stored in?

Can reshape arrays in many other ways

```python
>>> first = numpy.array([1, 2, 3, 4, 5, 6])
>>> first.shape
(6,)

>>> second = first.reshape(2, 3)
>>> second
array([[1, 2, 3],
       [4, 5, 6]])
```

Also aliases the data
New shape must have same size as old

```python
>>> first = numpy.zeros((2, 2))
>>> first.reshape(3, 3)
ValueError: total size of new array must be unchanged
```
Cannot possibly work because it is just creating an alias for the existing data

Change physical size using `array.resize`

```python
>>> block
array([[ 10,  20,  30],
       [110, 120, 130],
       [210, 220, 230]])
>>> block.resize(2, 2)
>>> block
array([[ 10,  20],
       [110, 120]])
```
Change physical size using `array.resize`

```python
>>> block
array([[ 10,  20,  30],
       [110, 120, 130],
       [210, 220, 230]])
>>> block.resize(2, 2)
>>> block
array([[ 10,  20],
       [110, 120]])
```

What happens when the array grows?

Review:
- Arrays are blocks of homogeneous data
- Most operations create aliases
- Can be reshaped (size remains the same)
- Or resized