Computational Programming with Python
Unit 6: More on arrays, more on functions

Numerical Analysis, Lund University
Lecturer: Claus Führer

2016
Solving a Linear System

If $A$ is a matrix and $b$ is a vector you solve the linear equation

$$A \cdot x = b$$

using `solve` which has the syntax $x = \text{solve}(A,b)$.

**Example**

We want to solve

\[
\begin{align*}
    x_1 + 2x_2 &= 1 \\
    3x_1 + 4x_2 &= 4
\end{align*}
\]

```python
from scipy.linalg import solve
A = array([[1., 2.], [3., 4.]])
b = array([1., 4.])
x = solve(A, b)
dot(A, x) # should be almost b
```
Slices are similar to that of lists and vectors *except* that there are now *two dimensions*.

- \( M[i,:] \) a *vector* filled by the row \( i \) of \( M \)
- \( M[:j] \) a *vector* filled by the column \( j \) of \( M \)
- \( M[2:4,:] \) slice 2:4 on the rows only
- \( M[2:4,1:4] \) slice on rows and columns

**Omitting a dimension**

If you omit an index or a slice, Scipy assumes you are taking *rows only*.

- \( M[3] \) is the third row of \( M \)
- \( M[1:3] \) is a matrix with the second and third rows of \( M \).
Altering a Matrix

You may alter a matrix using slices or direct access.

- \( M[2,3] = 2. \)
- \( M[2,:) = \text{<a vector>} \)
- \( M[1:3,:) = \text{<a matrix>} \)
- \( M[1:4,2:5] = \text{<a matrix>} \)

The matrices and vectors above must have the right size to “fit” in the matrix \( M \).
Some technical terms

**Rank**  The rank of an array is the number of indices used to identify an element. A matrix has rank 2, a vector has rank 1, a scalar has rank 0.

Don’t confound this term with the rank used in Linear Algebra.

**Shape**  The shape is a tuple with the dimensions of the array: A $2 \times 3$ matrix is represented in Python by an array with shape $(2, 3)$.

**Length**  The number of elements in an array. An array with shape $(2, 3)$ has length 6. An array with shape $(2, \_)$ (representing a vector) has length 2.

Don’t confound this term with the term length used in Linear Algebra.
Rank of matrix slices

When slicing the rank of the result object is as follows:

<table>
<thead>
<tr>
<th>access</th>
<th>rank</th>
<th>kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>index,index</td>
<td>0</td>
<td>scalar</td>
</tr>
<tr>
<td>slice,index</td>
<td>1</td>
<td>vector</td>
</tr>
<tr>
<td>slice,slice</td>
<td>2</td>
<td>matrix</td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>access</th>
<th>shape</th>
<th>rank</th>
<th>kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>M[:2,1:-1]</td>
<td>(2,2)</td>
<td>2</td>
<td>matrix</td>
</tr>
<tr>
<td>M[1,:)</td>
<td>4</td>
<td>1</td>
<td>vector</td>
</tr>
<tr>
<td>M[1,1]</td>
<td>Ø</td>
<td>0</td>
<td>scalar</td>
</tr>
<tr>
<td>M[1:2,:)</td>
<td>(1,4)</td>
<td>2</td>
<td>matrix</td>
</tr>
<tr>
<td>M[1:2,1:2]</td>
<td>(1,1)</td>
<td>2</td>
<td>matrix</td>
</tr>
</tbody>
</table>
Reshaping

From a given tensor (vector or matrix) one may obtain another tensor by *reshaping*. 
Reshaping Example

```python
A = arange(6)
A.reshape(1,6)
A.reshape(6,1)
A.reshape(2,3)
A.reshape(3,2)
```

```
0 1 2 3 4 5
0 1 2 3 4 5
0 1 2
0 1 2
0 1 2
0 1 2
0 1 2
```
Reshaping Trick

Note that Python can *guess one of the new dimensions*. Just give a negative integer for the dimension to be guessed:

```python
A = arange(12)  # a vector of length 12
A.reshape(1,-1)  # row matrix
A.reshape(-1,1)  # column matrix
A.reshape(3,-1)  # 3,4 matrix
A.reshape(-1,4)  # same
```
Building Matrices

- Piling vectors
- Stacking vectors
- Stacking column matrices

The universal method to build matrices is **concatenate**. This function is called by several convenient functions

- **hstack** to stack matrices *horizontally*
- **vstack** to stack matrices *vertically*
- **column_stack** to stack *vectors* in columns
Stacking Vectors

$v1 = array([1,2])$
$v2 = array([3,4])$

$vstack([v1,v2])$

$column_stack([v1,v2])$
sum, max, min

You may perform a number of operations on arrays, either on the whole array, or column-wise or row-wise. The most common are

- **max**
- **min**
- **sum**

### Example

```
1  2  3  4
5  6  7  8
```

A. `sum()`

```
36
```

A. `sum(axis=0)`

The result is a *vector*. 

```
1  2  3  4
5  6  7  8
```

The result is a *vector*. 

Boolean Arrays
Boolean Arrays

One may use *Boolean arrays* to create a “template” for modifying another array:

```python
B = array([[True, False],
            [False, True]]) # the template array
M = array([[2, 3],
            [1, 4]])  # the other array
M[B] = 0    # using the template
M # [[0, 3], [1, 0]]
M[B] = 10, 20
M # [[10, 3], [1, 20]]
```
Creating Boolean Arrays

It might be just as tedious to create the boolean array by hand than to change the array directly. There are however many methods to create Boolean arrays.

Any *logical operator* will create a Boolean array instead of a Boolean.

```python
M = array([[2, 3], [1, 4]])
M > 2  # array([[False, True], [False, True]])
M == 0 # array([[False, False], [False, False]])
N = array([[2, 3], [0, 0]])
M == N # array([[True, True], [False, False]])
...
```

This allows the elegant syntax:

```python
M[M>2] = 0
# all the elements > 2 are replaced by 0
```
Comparing Arrays

Note that because array comparison create Boolean arrays, one cannot compare arrays directly.

The solution is to use the methods `all` and `any`:

```python
A = array([[1, 2, 3, 4]])
B = array([[1, 2, 3, 3]])
A == B  # creates array([[True, True, True, False]])
(A == B).all()  # False
(A != B).any()  # True
```
Boolean Operations

For the same reason as before you cannot use and, or nor not on Boolean arrays! Use the following replacement operators instead:

<table>
<thead>
<tr>
<th>logic operator</th>
<th>replacement for Bool arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>A and B</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>A or B</td>
<td>A</td>
</tr>
<tr>
<td>not A</td>
<td>-A</td>
</tr>
</tbody>
</table>

```python
a = array([True, True, False, False])
b = array([True, False, True, False])
a and b  # error!
a & b  # array([True, False, False, False])
a | b  # array([True, Trues, True, False])
-a    # array([False, False, True, True])
```
Universal Functions
Universal Functions

Definition

A universal function (or ufunc) is a function that operates on arrays in an element-by-element fashion. That is, a ufunc is a “vectorized” wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs.

Examples:

```python
from scipy import *
sin
cos
exp
```
Vectorized Functions

Non-universal functions can be wrapped to behave like universal functions.
This is done by the command `vectorize`.

```python
# A non-universal function
def heaviside(x):
    if x >= 0:
        return 1.
    else:
        return 0.

heaviside(linspace(-1,1,100))  # returns an error

vheaviside=vectorize(heaviside)  # a new function
vheaviside(linspace(-1,1,100))  # does the job
```
Vectorized Functions  (Cont.)

```matlab
xvals = linspace(-1, 1, 100)
plot(xvals, vectorize(heaviside)(xvals))
axis([-1.5, 1.5, -0.5, 1.5])
```