A quick tour of numpy basics: arrays, plotting
Numpy: arrays and more

Numpy offers several fundamental structures for math...

- **array**: a list-like object.
  Unlike a list, it has a **fixed** size and must hold a numeric type (float etc.)
  Can be any number of dimensions!
- **matrix**: like a 2d array, but with more structure specific to matrices

```python
import numpy as np
x = np.array([1.0, 2.0, 3.0])
y = np.zeros((3,4))  # 3x4 array of zeros
z = np.linspace(0, 1, 110)  # [0, 0.01, 0.02, ..., 0.99, 1]
y.shape  # (3,4)
```

**Arrays:**

- Many ways to initialize (from a list, an array of zeros...)
- Useful: `linspace` (equally spaced points in an interval)
- `x.shape`: tuple of dimensions
- **Important**: In 2d, indexing uses tuples:

```python
a = np.array([[1, 2], [3, 4]])
print(a[1, 1])  # 4  (NOT a[1][1])
```
Numpy offers ‘vectorized’ functionality for most operations

- For arithmetic: done with overloaded $+,-,\times,/$

```python
x = np.array([1, 2, 3])
y = np.array([4, 3, 0])
z = x * y  # z is now [4, 6, 0]
```

- Vectorized: apply ‘to each element of’ an array (element-wise)
  Such functions construct a new result array and return it

- ‘typical’ math expressions (mostly) work, e.g. $z + 3\times x + 4\times y$

- Also works with $\text{max}$, $\text{sin}$, $\text{cos}$ etc:

```python
x = np.linspace(0, 1, 100)
y = np.sin(x)  # y[i] is sin(x[i])
maxval = np.max(y)
```

- Caution: $A\times x$ is not matrix multiplication!
  Use `np.dot(a, x)` or `a.dot(x)` instead ($\times$ is elementwise).
Slicing in numpy **is different than for python lists**. It is defined to work better with arrays/matrices of numbers.

- Slice notation in 1d is the same...

```python
x = np.array([1, 4, 9, 16, 25])
y = x[1:3]
print(y)  # prints [4, 9]
```

- The usual ‘blank’ notation applies (e.g. `a[1:]` for 1 to end).

- However, numpy slices act as **references** to the data (not copies!)

- slices are ‘windows’ into the data that see a subset

```python
x = np.array([1, 4, 9, 16, 25])
y = x[1:3]
y[1] = 77
# now x is [1, 77, 9, 16, 25]
```

```python
x = [1, 4, 9, 16, 25]
y = x[1:3]
y[1] = 77
# now x is [1, 4, 9, 16, 25]
```
• slices work in 2d (unlike python lists):

```python
a = [[1,2,3], [4,5,6], [7,8,9]]
sub = a[0:2,0:2]  # sub sees [[1,2],[4,5]]
```

• You can set subsets of an array with slices (just as with lists):

```python
a = [[1,2,3], [4,5,6], [7,8,9]]
b = [[[10,0], [0,10]]
a[1:3,1:3] = b
# now a is [[1,2,3], [4,10,0], [7,0,10]]
b[1] = 77  # a unchanged (a and b are not linked!)
```

```python
a[2,:]  # row 2 of a
a[:,2]  # col 2 of a
```

• This sets the specified elements to the **values** given on the RHS (so values are copied from the RHS to the LHS data).
Plotting

Plotting can be done through `matplotlib.pyplot`.

- Syntax closely mimics Matlab
- You can give data as numpy arrays or lists (mostly)

```python
import numpy as np
import numpy.matplotlib as plt
x = np.linspace(0, 2, 1000)
y = x**3 - x  # vectorized!
plt.figure()  # define new figure window
plt.plot(x, y)
plt.show()
```

- `show()` tells python to render the plot (so you can see it)
- Figures can be given ids (`figure(1)`, etc.) to make more than one
- plots ‘hold’ by default: new plot commands add to the existing plot

```python
... 
plt.plot(x, y)
y2 = np.sin(x)
plt.plot(x, y2)
```
Plotting: decorations and saving

• Axis labels, titles...
  
  ```python
  plt.xlabel('x')
  plt.ylabel('y')
  plt.title('text above the plot')
  ```

• You can also add legends [see documentation for pyplot]

• Line styles (dashed etc.) and colors:
  
  ```python
  plt.plot(x, y, '-k', x, y2, '--r')  # black, solid and red, dashed
  plt.plot(x, y, '.b', markersize=40)  # blue, dots only, size 40
  ```

Saving plots:

• Set figure output size (in inches)
  
  ```python
  use plt.figure(..., figsize=(m,n)) for $m \times n$ inches
  ```

• Save (as .pdf, .eps or .png) using
  
  ```python
  plt.savefig(filename)
  # ..or to remove white border...
  plt.savefig(filename, bbox_inches='tight')
  ```