Introductory Scientific Computing with Python

More plotting, lists and numpy arrays

FOSSEE

Department of Aerospace Engineering
IIT Bombay

SciPy India, 2015
December, 2015

Outline

Outline

Why would I plot f(x)?

Do we plot analytical functions or experimental data?

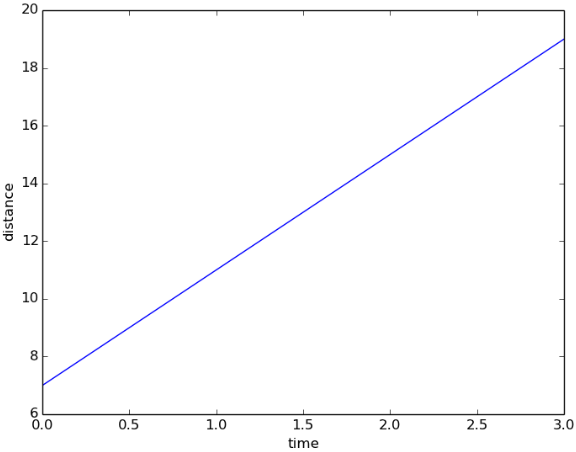
```
In []: time = [0., 1., 2, 3]

In []: distance = [7., 11, 15, 19]

In []: plot(time,distance)
Out[]: [<matplotlib.lines.Line2D object at 0xa73a

In []: xlabel('time')
Out[]: <matplotlib.text.Text object at 0x986e9ac>

In []: ylabel('distance')
Out[]: <matplotlib.text.Text object at 0x98746ec>
```



Is this what you have?

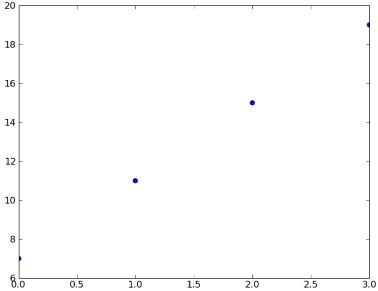
Plotting points

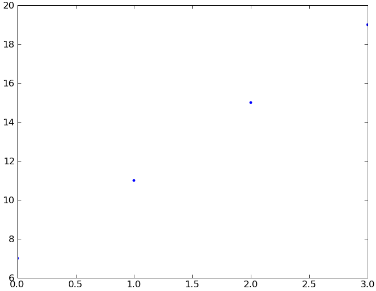
- What if we want to plot the points?

```
In []: clf()

In []: plot(time, distance, 'o')
Out[]: [<matplotlib.lines.Line2D object

In []: clf()
In []: plot(time, distance, '.')
Out[]: [<matplotlib.lines.Line2D object
```


20
18
16
14
12
10
8
6
0.0
0.5
1.0
1.5
2.0
2.5
3.0


20
18
16
14
12
10
8
6
0.0
0.5
1.0
1.5
2.0
2.5
3.0

Additional Line Styles

- `'o'` - Filled circles
- `'.'` - Small Dots
- `'-'` - Lines
- `'--'` - Dashed lines

Outline

Lists: Introduction

```
In []: time = [0, 1, 2, 3]

In []: distance = [7, 11, 15, 19]
```

What are `x` and `y`?

lists!!

Lists: Initializing & accessing elements

```
In []: mtlist = []
```

Empty List

```
In []: p = [ 2, 3, 5, 7]

In []: p[1]
Out[]: 3

In []: p[0]+p[1]+p[-1]
Out[]: 12
```

List: Slicing

Remember...

```
In []:  p = [ 2, 3, 5, 7]
```

```
In []: p[1:3]
Out[]: [3, 5]
```

A slice

```
In []: p[0:-1]
Out[]: [2, 3, 5]
In []: p[1:]
Out[]: [3, 5, 7]
```

List: Slicing . . .

```
In []: p[0:4:2]
Out[]: [2, 5]
In []: p[0::2]
Out[]: [2, 5]
In []: p[::2]
Out[]: [2, 5]
In []: p[::3]
Out[]: [2, 7]
In []: p[::-1]
Out[]: [7, 5, 3, 2]
```

```
list[initial:final:step]
```

List: Slicing

What is the output of the following?

```
In []: p[1::2]

In []: p[1:-1:2]
```

List operations

```
In []: b = [ 11, 13, 17]
In []: c = p + b

In []: c
Out[]: [2, 3, 5, 7, 11, 13, 17]

In []: p.append(11)
In []: p
Out[]: [ 2, 3, 5, 7, 11]
```

Question: Does `c` change now that `p` is changed? 10 m

Outline

Simple Pendulum - L and T

Let us look at the Simple Pendulum experiment.

L	T	T^2
0.1	0.69	
0.2	0.90	
0.3	1.19	
0.4	1.30	
0.5	1.47	
0.6	1.58	
0.7	1.77	
0.8	1.83	
0.9	1.94	

$$T \approx 2\pi\sqrt{L/g}$$

$$L \propto T^2$$

Let's use lists

```
In []: L = [0.1, 0.2, 0.3, 0.4, 0.5,
             0.6, 0.7, 0.8, 0.9]

In []: t = [0.69, 0.90, 1.19,
             1.30, 1.47, 1.58,
             1.77, 1.83, 1.94]
```

Gotcha: Make sure `L` and `t` have the same number of elements

```
In []: print(len(L), len(t))
```

Plotting L vs T^2

- We must square each of the values in `t`
- How do we do it?
- We use a `for` loop to iterate over `t`

Plotting L vs T^2

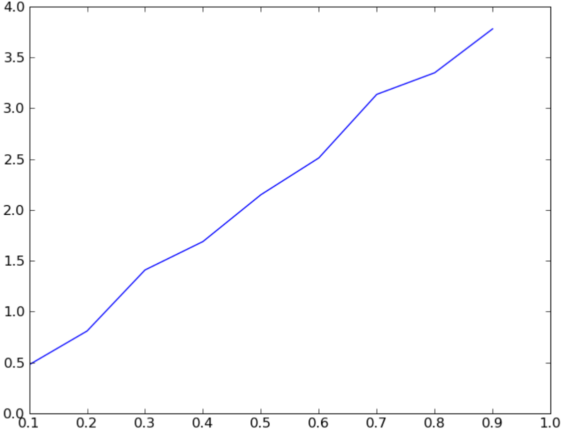
```
In []: tsq = []

In []: for time in t:
 ....:     tsq.append(time*time)
 ....:
 ....:
```

This gives `tsq` which is the list of squares of `t` values.

```
In []: print(len(L), len(t), len(tsq))
Out[]: 9 9 9

In []: plot(L, tsq)
```


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

This seems tedious

- Lists
  - Nice
  - Not too convenient
  - Slow
- Enter NumPy arrays
  - Fixed size, data type
  - Fast
  - Very convenient

Outline

NumPy arrays

```
In []: t = array(t)

In []: tsq = t*t

In []: print(tsq)

In []: plot(L, tsq) # works!
```

Speed?

Lets use range to create a large list.

```
In []: t = range(1000000)
In []: tsq = []

In []: for time in t:
 ....:     tsq.append(time*time)
 ....:
 ....:
```

Now try it with

```
In []: t = array(t)

In []: tsq = t*t
```

How fast is this?

Lets define a function for the list

```
In []: def sqr(arr):
   ...:     result = []
   ...:     for x in arr:
   ...:         result.append(x*x)
   ...:     return result
   ...:

In []: tsq = sqr(t)
```

Aside: Defining functions

- Consider the function `f(x) = x^2`
- Let's write a Python function, equivalent to this

```
In[]: def f(x):
....:     return x*x
....:

In[]: f(1)
In[]: f(2)
```

- `def` is a keyword
- `f` is the name of the function
- `x` the parameter of the function
- `return` is a keyword

IPython tip: Timing

Try the following:

```
In []: %timeit sqr(t)

In []: %timeit?
```

- **`%timeit`**: accurate, many measurements
- Can also use **`%time`**
- **`%time`**: less accurate, one measurement

25 m

Exercise

Find out the speed difference between the **`sqr`** function and **`t*t`** on the numpy array.

The numpy module

- Efficient, powerful array type
- Abstracts out standard operations on arrays
- Convenience functions
- **ipython -pylab** imports part of numpy
- Without the Pylab mode do:

```
In []: import numpy

In []: from numpy import *
```

numpy arrays

- Fixed size (**`arr.size`**)
- Same type (**`arr.dtype`**)
- Arbitrary dimensionality: **`arr.shape`**
- **`shape`**: extent (size) along each dimension
- **`arr.itemsize`**: number of bytes per element
- Note: **`shape`** can change so long as the **`size`** is constant
- Indices start from 0
- Negative indices work like lists

numpy arrays

```
In []: a = array([1,2,3,4])
In []: b = array([2,3,4,5])

In []: print(a[0], a[-1])
1, 4

In []: a[0] = -1
In []: a[0] = 1
```

Operations are elementwise

Simple operations

```
In []: a + b
Out[]: array([3, 5, 7, 9])
In []: a*b
Out[]: array([2, 6, 12, 20])
In []: a/b
Out[]: array([0, 0, 0, 0])
```

Operations are elementwise, types matter.

Data type matters

Try again with this:

```
In []: a = array([1.,2,3,4])
In []: a/b
```

Examples

`pi` and `e` are defined.

```
In []: x = linspace(0.0, 10.0, 200)
In []: x *= 2*pi/10
# apply functions to array.
In []: y = sin(x)
In []: y = cos(x)
In []: x[0] = -1
In []: print(x[0], x[-1])
-1.0 10.0
```

size, shape, rank etc.

```
In []: x = array([1., 2, 3, 4])
In []: size(x)
Out[]: 4
In []: x.dtype
dtype('float64')
In []: x.shape
Out[] (4,)
In []: rank(x)
Out[]: 1
In []: x.itemsize
Out[]: 8
```

Multi-dimensional arrays

```
In []: a = array([[ 0,  1,  2,  3],
   ...:             [10,11,12,13]])
In []: a.shape # (rows, columns)
Out[]: (2, 4)

In []: a[1,3]
Out[]: 13

In []: a[1,3] = -1
In []: a[1] # The second row
array([10,11,12,-1])
In []: a[1] = 0 # Entire row to zero.
```

Slicing arrays

```
In []: a = array([[1,2,3], [4,5,6],
   ...:            [7,8,9]])
In []: a[0,1:3]
Out[]: array([2, 3])
In []: a[1:,1:]
Out[]: array([[5, 6],
              [8, 9]])
In []: a[:,2]
Out[]: array([3, 6, 9])
In []: a[0::2,0::2] # Striding...
Out[]: array([[1, 3],
              [7, 9]])
# Slices refer to the same memory!
```

Array creation functions

- `array(object)`
- `linspace(start, stop, num=50)`
- `ones(shape)`
- `zeros((d1,...,dn))`
- `empty((d1,...,dn))`
- `identity(n)`
- `ones_like(x)`, `zeros_like(x)`, `empty_like(x)`

May pass an optional **`dtype=`** keyword argument
For more dtypes see: **`numpy.typeDict`**

Creation examples

```
In []: a = array([1,2,3], dtype=float)
In []: ones( (2, 3) )
Out[]: array([[ 1.,  1.,  1.],
               [ 1.,  1.,  1.]])
In []: identity(3)
Out[]: array([[ 1.,  0.,  0.],
               [ 0.,  1.,  0.],
               [ 0.,  0.,  1.]])
In []: ones_like(a)
Out[]: array([ 1.,  1.,  1.,  1.])
```

Array math

- Basic elementwise math (given two arrays `a, b`):
  - `a + b` → `add(a, b)`
  - `a - b`, → `subtract(a, b)`
  - `a * b`, → `multiply(a, b)`
  - `a / b`, → `divide(a, b)`
  - `a % b`, → `remainder(a, b)`
  - `a ** b`, → `power(a, b)`
- Inplace operators: `a += b`, or `add(a, b, a)`
  What happens if `a` is `int` and `b` is `float`?
- Logical operations: `==, !=, <, >`, etc.
- `sin(x), arcsin(x), sinh(x), exp(x), sqrt(x)` etc.
- `sum(x, axis=0), product(x, axis=0)`
- `dot(a, b)`

Convenience functions: loadtxt

- `loadtxt(file_name)`: loads a text file
- `loadtxt(file_name, unpack=True)`: loads a text file and unpacks columns

```
In []: x = loadtxt('pendulum.txt')
In []: x.shape
Out[]: (90, 2)

In []: x, y = loadtxt('pendulum.txt',
   ...:                unpack=True)
In []: x.shape
Out[]: (90,)
```

45 m

Advanced

- Only scratched the surface of `numpy`
- **`reduce`**, **`outer`**
- Typecasting
- More functions: **`take`**, **`choose`**, **`where`**, **`compress`**, **`concatenate`**
- Array broadcasting and **`None`**
- Record arrays

Learn more

- http://wiki.scipy.org/Tentative_NumPy_Tutorial
- http://numpy.org

Recap

- Basic concepts: creation, access, operations
- 1D, multi-dimensional
- Slicing
- Array creation, dtypes
- Math
- `loadtxt`

Example: plotting data from file

Data is usually present in a file!

Lets look at the **pendulum.txt** file.

```
In []: cat pendulum.txt
1.0000e-01 6.9004e-01
1.1000e-01 6.9497e-01
1.2000e-01 7.4252e-01
1.3000e-01 7.5360e-01

...
```

Reading `pendulum.txt`

- File contains L vs. T values
- First Column - L values
- Second Column - T values
- Let us generate a plot from the data file

Gotcha and an aside

Ensure you are in the same directory as
pendulum.txt
if not, do the following on IPython:

```
In []: %cd directory_containing_file
# Check if pendulum.txt is there.
In []: ls
# Also try
In []: !ls
```

Note: **%cd** is an IPython magic command. For more information do:

```
In []: ?
```

Exercise

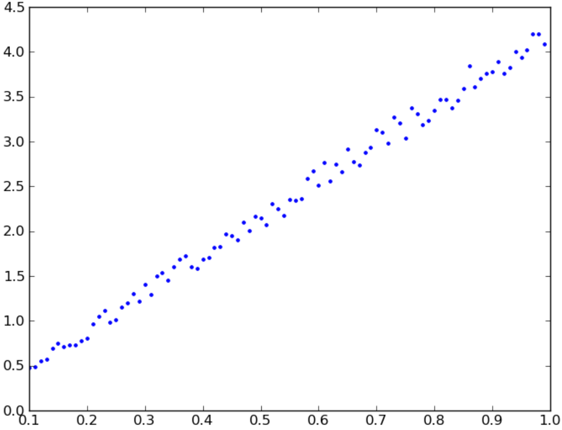
- Plot L versus T square with dots
- No line connecting points

Solution

```
In []: L, t = loadtxt('pendulum.txt',
 ....:                   unpack=True)
In []: plot(L, t*t, '.')
```

or

```
In []: x = loadtxt('pendulum.txt')
In []: L, t = x[:,0], x[:,1]
In []: plot(L, t*t, '.')
```


4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

Odds and ends

```
In []: mean(L)
Out[]: 0.54499999999999993

In []: std(L)
Out[]: 0.25979158313283879
```

Outline

1. Plotting Points
2. Lists
3. Simple Pendulum
   - `numpy` arrays
4. Summary

What did we learn?

- Plot attributes and plotting points
- Lists
- Introduction to `numpy` arrays

55 m