Advanced Python

FOSSEE

August 18, 2016

Outline

Outline

First Plot

- Start IPython with `-pylab`

```
$ ipython -pylab
```

```
In[]: p = linspace(-pi,pi,100)
In[]: plot(p, cos(p))
```

linspace

- `p` has a hundred points in the range -pi to pi

```
In[]: print p[0], p[-1], len(p)
```

- Look at the doc-string of `linspace` for more details

```
In[]: linspace?
```

- `plot` simply plots the two arguments with default properties

Outline

Plot color and thickness

```
In[]: clf()
In[]: plot(p, sin(p), 'r')
```

- Gives a sine curve in Red.

```
In[]: plot(p, cos(p), linewidth=2)
```

- Sets line thickness to 2

```
In[]: clf()
In[]: plot(p, sin(p), '.')
```

- Produces a plot with only points

```
In[]: plot?
```

title

```
In[]: x = linspace(-2, 4, 50)
In[]: plot(x, -x*x + 4*x - 5, 'r',
      linewidth=2)
In[]: title("Parabolic function -x^2+4x-5")
```

- We can set title using LaTeX

```
In[]: title("Parabolic function $-x^2+4x-5$")
```

Axes labels

```
In[]: xlabel("x")
In[]: ylabel("f(x)")
```

- We could, if required use LaTeX

Annotate

```
In[]: annotate("local maxima", xy=(2, -1))
```

- First argument is the annotation text
- The argument to `xy` is a tuple that gives the location of the text.

Limits of Plot area

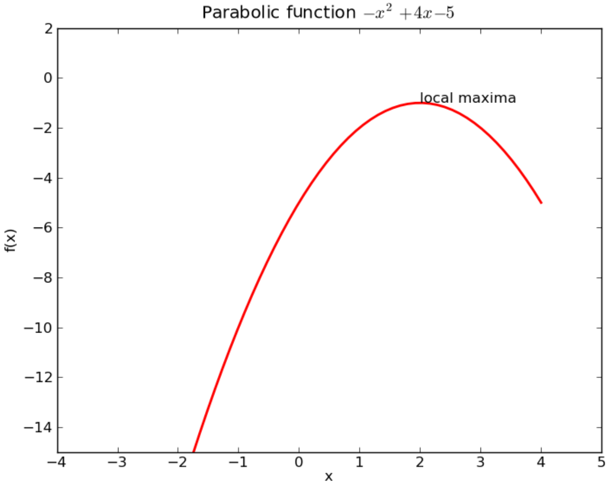
```
In[]: xlim()
In[]: ylim()
```

- With no arguments, `xlim` & `ylim` get the current limits
- New limits are set, when arguments are passed to them

```
In[]: xlim(-4, 5)
```

```
In[]: ylim(-15, 2)
```

Plot


Parabolic function $-x^2 + 4x - 5$
local maxima
f(x)
x
2
0
−2
−4
−6
−8
−10
−12
−14
−4
−3
−2
−1
0
1
2
3
4
5

Outline

Command history

- To see the history of commands, we typed

```
In[]: %hist
```

- All commands, valid or invalid, appear in the history
- `%hist` is a magic command, available only in IPython

```
In[]: %hist 5
 # last 5 commands

In[]: %hist 5 10
 # commands between 5 and 10
```

Saving to a script

- We wish to save commands for reproducing the parabola
- Look at the history and identify the commands that will reproduce the parabolic function along with all embellishment
- `%save` magic command to save the commands to a file

```
In[]: %save plot_script.py 1 3-6 8
```

- File name must have a `.py` extension

Running the script

```
In[]: %run -i plot_script.py
```

- There were no errors in the plot, but we don't see it!
- Running the script means, we are not in interactive mode
- We need to explicitly ask for the image to be shown

```
In[]: show()
```

- `-i` asks the interpreter to check for names, unavailable in the script, in the interpreter
- `sin`, `plot`, etc. are taken from the interpreter

Outline

savefig

```
In[]: x = linspace(-3*pi,3*pi,100)
In[]: plot(x,sin(x))
In[]: savefig('sine.png')
```

- `savefig` takes one argument
- The file-type is decided based on the extension
- `savefig` can save as png, pdf, ps, eps, svg

Outline

Overlaid plots

```
In[]: x = linspace(0, 50, 10)
In[]: plot(x, sin(x))
```

- The curve isn't as smooth as we expected
- We chose too few points in the interval

```
In[]: y = linspace(0, 50, 500)
In[]: plot(y, sin(y))
```

- The plots are overlaid
- It is the default behaviour of `pylab`

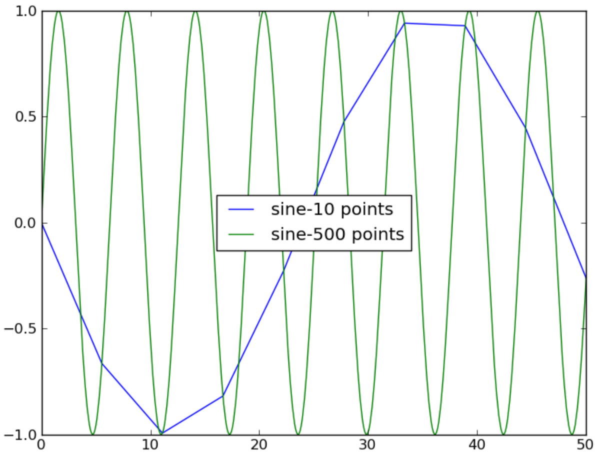
Legend

```
In[]: legend(['sine-10 points',
       'sine-500 points'])
```

- Placed in the location, `pylab` thinks is 'best'
- `loc` parameter allows to change the location

```
In[]: legend(['sine-10 points',
      'sine-500 points'],
         loc='center')
```

Overlaid Plots



Plotting in separate figures

```
In[]: clf()
In[]: x = linspace(0, 50, 500)
In[]: figure(1)
In[]: plot(x, sin(x), 'b')
In[]: figure(2)
In[]: plot(x, cos(x), 'g')
```

- `figure` command allows us to have plots separately
- It is also used to switch context between the plots

```
In[]: savefig('cosine.png')
In[]: figure(1)
In[]: title('sin(y)')
In[]: savefig('sine.png')
In[]: close()
In[]: close()
```

Subplots

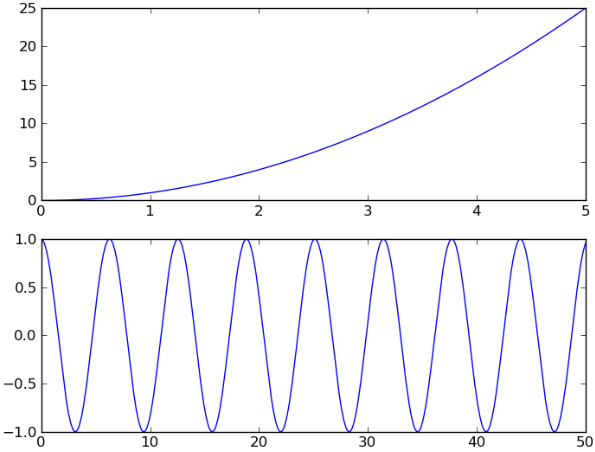
```
In[]: subplot(2, 1, 1)
```

- number of rows
- number of columns
- plot number, in serial order, to access or create

```
In[]: subplot(2, 1, 2)
In[]: x = linspace(0, 50, 500)
In[]: plot(x, cos(x))

In[]: subplot(2, 1, 1)
In[]: y = linspace(0, 5, 100)
In[]: plot(y, y ** 2)
```

Subplots



Outline

Loading data

- `primes.txt` contains a list of primes listed column-wise
- We read the data using `loadtxt`

```
In[]: primes = loadtxt('primes.txt')
In[]: print primes
```

- `primes` is a sequence of floats

Reading two column data

- `pendulum.txt` has two columns of data
- Length of pendulum in the first column
- Corresponding time period in second column
- `loadtxt` requires both columns to be of same length

```
In[]: pend = loadtxt('pendulum.txt')
In[]: print pend
```

- `pend` is not a simple sequence like `primes`

Unpacking with `loadtxt`

```
In[]: L, T = loadtxt('pendulum.txt',
                     unpack=True)
In[]: print L
In[]: print T
```

- We wish to plot L vs. T^2
- `square` function gives us the squares
- (We could instead iterate over T and calculate)

```
In[]: Tsq = square(T)

In[]: plot(L, Tsq, '.')
```

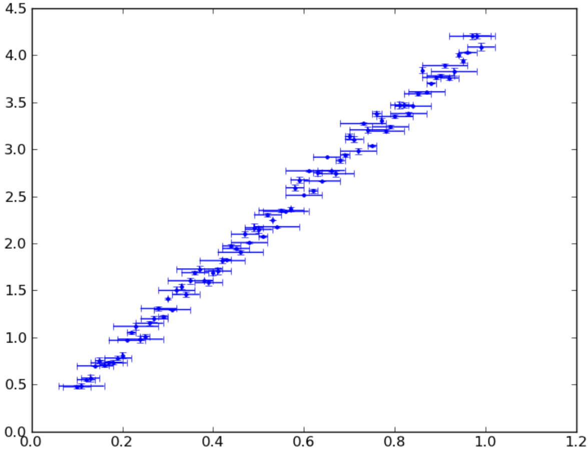
errorbar

- Experimental data always has errors
- `pendulum_error.txt` contains errors in L and T
- Read the values and make an error bar plot

```
In[]:  L, T, L_err, T_err = \
       loadtxt('pendulum_error.txt',
       unpack=True)
In[]:  Tsq = square(T)

In[]:  errorbar(L, Tsq , xerr=L_err,
       yerr=T_err, fmt='b.')
```

Errorbar



Outline

Scatter Plot

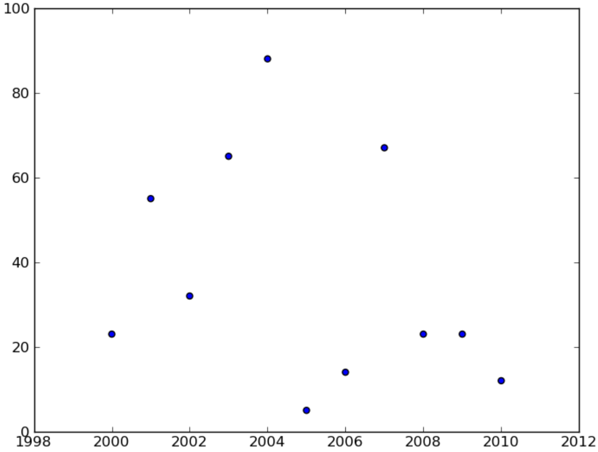
- The data is displayed as a collection of points
- Value of one variable determines position along x-axis
- Value of other variable determines position along y-axis
- Let's plot the data of profits of a company

```
In[]:  year, profit = loadtxt(
                          'company-a-data.txt',
                          dtype=type(int()))

In[]: scatter(year, profit)
```

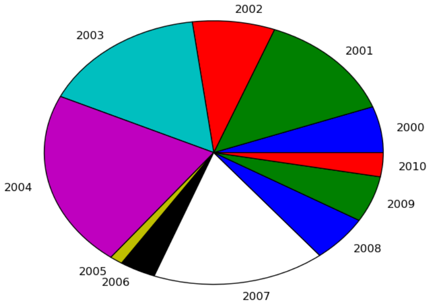
- `dtype=int`; default is float

Scatter Plot



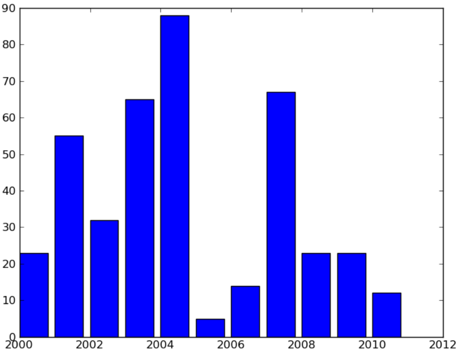
Pie Chart

```
In[]: pie(profit, labels=year)
```



Bar Chart

```
In[]: bar(year, profit)
```



Log-log plot

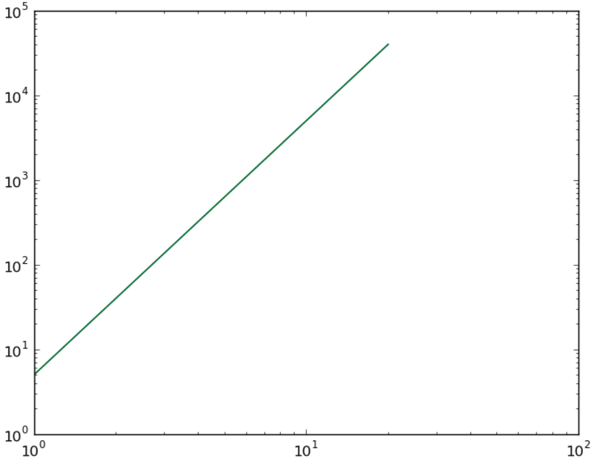
- Plot a `log-log` chart of $y = 5x^3$ for x from 1 to 20

```
In[]: x = linspace(1,20,100)
In[]: y = 5*x**3

In[]: loglog(x, y)
In[]: plot(x, y)
```

- Look at `http://matplotlib.sourceforge.net/contents.html` for more!

Log-log plot Plot



Outline

Arrays: Introduction

- Similar to lists, but homogeneous
- Much faster than arrays

```
In[]: a1 = array([1,2,3,4])
In[]: a1 # 1-D
In[]: a2 = array([[1,2,3,4],[5,6,7,8]])
In[]: a2 # 2-D
```

arange and shape

```
In[]: ar1 = arange(1, 5)
In[]: ar2 = arange(1, 9)
In[]: print ar2
In[]: ar2.shape = 2, 4
In[]: print ar2
```

- `linspace` and `loadtxt` also returned arrays

```
In[]: ar1.shape
In[]: ar2.shape
```

Special methods

```
In[]: identity(3)
```

- array of shape (3, 3) with diagonals as 1s, rest 0s

```
In[]: zeros((4,5))
```

- array of shape (4, 5) with all 0s

```
In[]: a = zeros_like([1.5, 1, 2, 3])
In[]: print a, a.dtype
```

- An array with all 0s, with similar shape and dtype as argument
- Homogeneity makes the dtype of a to be float
- `ones, ones_like, empty, empty_like`

Operations on arrays

```
In[]:  a1
In[]:  a1 * 2
In[]:  a1
```

- The array is not changed; New array is returned

```
In[]: a1 + 3
In[]: a1 - 7
In[]: a1 / 2.0
```

Operations on arrays . . .

- Like lists, we can assign the new array, the old name

```
In[]: a1 = a1 + 2
In[]: a1
```

- Beware of Augmented assignment!

```
In[]: a, b = arange(1, 5), arange(1, 5)
In[]: print a, a.dtype, b, b.dtype
In[]: a = a/2.0
In[]: b /= 2.0
In[]: print a, a.dtype, b, b.dtype
```

- Operations on two arrays; element-wise

```
In[]: a1 + a1
In[]: a1 * a2
```

Outline

Accessing & changing elements

```
In[]:  A = array([12, 23, 34, 45, 56])

In[]:  C = array([[11, 12, 13, 14, 15],
                  [21, 22, 23, 24, 25],
                  [31, 32, 33, 34, 35],
                  [41, 42, 43, 44, 45],
                  [51, 52, 53, 54, 55]])
```

```
In[]: A[2]
In[]: C[2, 3]
```

- Indexing starts from 0
- Assign new values, to change elements

```
In[]: A[2] = -34
In[]: C[2, 3] = -34
```

Accessing rows

- Indexing works just like with lists

```
In[]: C[2]
In[]: C[4]
In[]: C[-1]
```

- Change the last row into all zeros

```
In[]:  C[-1] = [0, 0, 0, 0, 0]
```

OR

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

Accessing columns

```
In[]:  C[:, 2]
In[]:  C[:, 4]
In[]:  C[:, -1]
```

- The first parameter is replaced by a `:` to specify we require all elements of that dimension

```
In[]: C[:, -1] = 0
```

Slicing

```
In[]: I = imread('squares.png')
In[]: imshow(I)
```

- The image is just an array

```
In[]: print I, I.shape
```

1. Get the top left quadrant of the image
2. Obtain the square in the center of the image

Slicing . . .

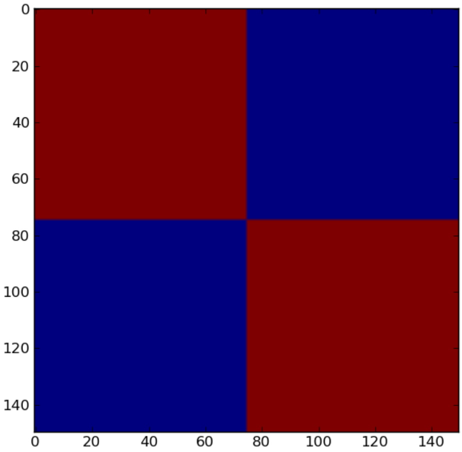
- Slicing works just like with lists

```
In[]: C[0:3, 2]
In[]: C[2, 0:3]
In[]: C[2, :3]

In[]: imshow(I[:150, :150])

In[]: imshow(I[75:225, 75:225])
```

Image after slicing



Striding

- Compress the image to a fourth, by dropping alternate rows and columns
- We shall use striding
- The idea is similar to striding in lists

```
In[]: C[0:5:2, 0:5:2]
In[]: C[::2, ::2]
In[]: C[1::2, ::2]
```

- Now, the image can be shrunk by

```
In[]: imshow(I[::2, ::2])
```

Outline

Matrix Operations using `arrays`

We can perform various matrix operations on `arrays`
A few are listed below.

Operation	How?	Example
Transpose	`.T`	`A.T`
Product	`dot`	`dot(A, B)`
Inverse	`inv`	`inv(A)`
Determinant	`det`	`det(A)`
Sum of all elements	`sum`	`sum(A)`
Eigenvalues	`eigvals`	`eigvals(A)`
Eigenvalues & Eigenvectors	`eig`	`eig(A)`
Norms	`norm`	`norm(A)`
SVD	`svd`	`svd(A)`

Outline

Least Square Fit

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

- Both the plots, aren't what we expect – linear plot
- Enter Least square fit!

Matrix Formulation

- We need to fit a line through points for the equation $T^2 = m \cdot L + c$
- In matrix form, the equation can be represented as $T_{sq} = A \cdot p$, where T_{sq} is $\begin{bmatrix} T_1^2 \\ T_2^2 \\ \vdots \\ T_N^2 \end{bmatrix}$, A is $\begin{bmatrix} L_1 & 1 \\ L_2 & 1 \\ \vdots & \vdots \\ L_N & 1 \end{bmatrix}$ and p is $\begin{bmatrix} m \\ c \end{bmatrix}$
- We need to find p to plot the line

Least Square Fit Line

```
In[]:  A = array((L, ones_like(L)))
In[]:  A.T
In[]:  A
```

- We now have `A` and `tsq`

```
In[]: result = lstsq(A, tsq)
```

- Result has a lot of values along with m and c, that we need

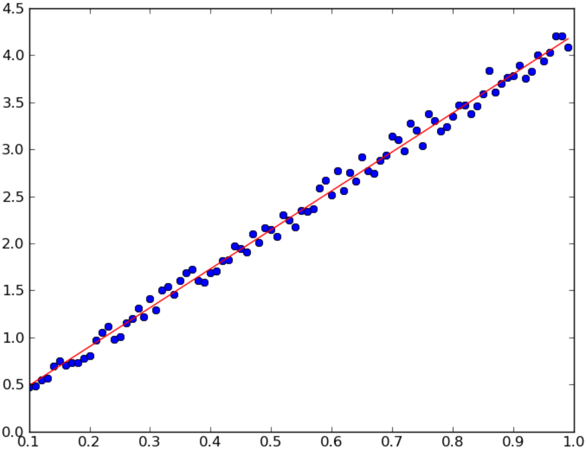
```
In[]: m, c = result[0]
In[]: print m, c
```

Least Square Fit Line

- Now that we have m and c, we use them to generate line and plot

```
In[]:  tsq_fit = m * L + c
In[]:  plot(L, tsq, 'bo')
In[]:  plot(L, tsq_fit, 'r')
```

Least Square Fit Line



Outline

Solution of linear equations

Consider,

$$3x + 2y - z = 1$$
$$2x - 2y + 4z = -2$$
$$-x + \frac{1}{2}y - z = 0$$

Solution:

$$x = 1$$
$$y = -2$$
$$z = -2$$

Solving using Matrices

Let us now look at how to solve this using `matrices`

```
In []: A = array([[3,2,-1],
                  [2,-2,4],
                  [-1, 0.5, -1]])
In []: b = array([1, -2, 0])
In []: x = solve(A, b)
```

Solution:

```
In []: x
Out[]: array([ 1., -2., -2.])
```

Let's check!

```
In []: Ax = dot(A, x)
In []: Ax
Out[]: array([ 1.00000000e+00,  -2.00000000e+00,
               -1.11022302e-16])
```

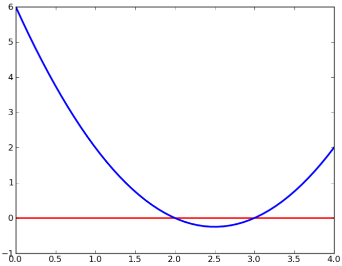
The last term in the matrix is actually 0!
We can use `allclose()` to check.

```
In []: allclose(Ax, b)
Out[]: True
```

`roots` of polynomials

- `roots` function can find roots of polynomials
- To calculate the roots of $x^2 - 5x + 6$

```
In []: coeffs = [1, -5, 6]
In []: roots(coeffs)
Out[]: array([3., 2.])
```



SciPy: `fsolve`

```
In []: from scipy.optimize import fsolve
```

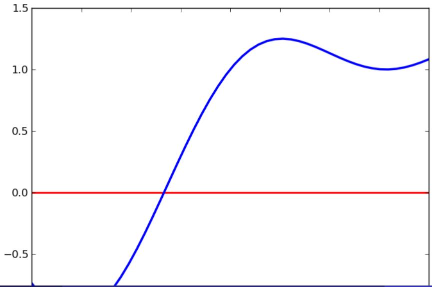
- Finds the roots of a system of non-linear equations
- Input arguments - Function and initial estimate
- Returns the solution

fsolve ...

Find the root of $sin(z) + cos^2(z)$ nearest to 0

```
In []: def g(z):
 ....:     return sin(z)+cos(z)*cos(z)

In []: fsolve(g, 0)
Out[]: -0.66623943249251527
```



Outline

Solving ODEs using SciPy

- Consider the spread of an epidemic in a population
- $\frac{dy}{dt} = ky(L - y)$ gives the spread of the disease
- L is the total population.
- Use $L = 2.5E5, k = 3E - 5, y(0) = 250$
- Define a function as below

```
In []: from scipy.integrate import odeint
In []: def epid(y, t):
  ....      k = 3.0e-5
  ....      L = 2.5e5
  ....      return k*y*(L-y)
  ....
```

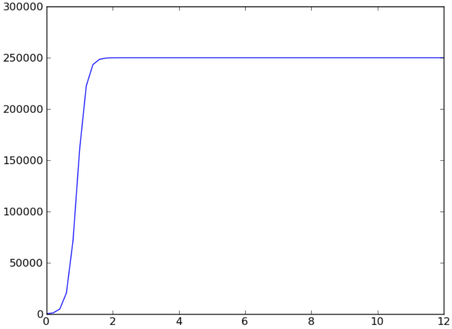
Solving ODEs using SciPy . . .

```
In []: t = linspace(0, 12, 61)

In []: y = odeint(epid, 250, t)

In []: plot(t, y)
```

Result



ODEs - Simple Pendulum

We shall use the simple ODE of a simple pendulum.

$$\ddot{\theta} = -\frac{g}{L} sin(\theta)$$

- This equation can be written as a system of two first order ODEs

$$\dot{\theta} = \omega \quad (1)$$

$$\dot{\omega} = -\frac{g}{L} sin(\theta) \quad (2)$$

$$\text{At } t = 0 :$$

$$\theta = \theta_0(10^o) \quad \& \quad \omega = 0 \; (Initial\ values)$$

ODEs - Simple Pendulum . . .

- Use `odeint` to do the integration

```
In []: def pend_int(initial, t):
  ....     theta = initial[0]
  ....     omega = initial[1]
  ....     g = 9.81
  ....     L = 0.2
  ....     F=[omega, -(g/L)*sin(theta)]
  ....     return F
  ....
```

ODEs - Simple Pendulum . . .

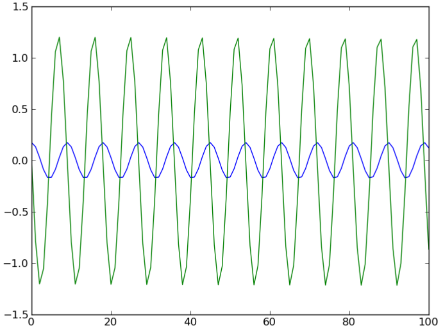
- `t` is the time variable
- `initial` has the initial values

```
In []: t = linspace(0, 20, 101)
In []: initial = [10*2*pi/360, 0]
```

ODEs - Simple Pendulum . . .

```
In []: from scipy.integrate import odeint
In []: pend_sol = odeint(pend_int,
                         initial,t)
```

Result



Outline

hello.py

- Script to print ‘hello world’ – `hello.py`

```
print "Hello world!"
```

- We have been running scripts from IPython

```
In[]: %run -i hello.py
```

- Now, we run from the shell using python

```
$ python hello.py
```

Simple plot

- Save the following in `sine_plot.py`

```
x = linspace(-2*pi, 2*pi, 100)
plot(x, sin(x))
show()
```

- Now, let us run the script

```
$ python sine_plot.py
```

- What's wrong?

Importing

- `-pylab` is importing a lot of functionality
- Add the following to the top of your file

```
from scipy import *
```

```
$ python sine_plot.py
```

- Now, plot is not found
- Add the following as the second line of your script

```
from pylab import *
```

```
$ python sine_plot.py
```

- It works!

Importing ...

- $\star$ imports everything from `scipy` and `pylab`
- But, It imports lot of unnecessary stuff
- And two modules may contain the same name, causing a conflict
- There are two ways out

```
from scipy import linspace, pi, sin
from pylab import plot, show
```

- OR change the imports to following and
- Replace `pi` with `scipy.pi`, etc.

```
import scipy
import pylab
```

Outline

Very useful packages

- http://ipython.org/
- http://sympy.org/
- http://pandas.pydata.org/