Using Python for Scientific Computing
Session 3 - NumPy, SciPy, Matplotlib

Felix Steffenhagen

University of Freiburg

May 4, 2011
Inhalt

1 NumPy

2 SciPy

3 Plotting and Data Visualization
What is NumPy?

- Fundamental package for scientific computing in Python
- Provides multidimensional arrays, matrices and polynom objects
- Fast operations on arrays through vectorized functions
- Differences to Python sequences:
  - Fixed size at creation
  - All elements of the same data type
  - Greater variety on numerical datatypes (e.g. int8, int32, uint32, float64)
  - Highly efficient (implemented in C)
- Base for many other scientific related packages
Python is slow(er) ...

Simple test: Multiply two arrays of length 10,000,000

**pure Python**

```python
import time
l = 10000000
start = time.time()
a, b = range(l), range(l)
c = []
for i in a:
    c.append(a[i] * b[i])
t = time.time() - start
print("Duration: %s" % t)
```

Duration: 4.67 s

**Using numpy**

```python
import numpy as np
import time
l = 10000000
start = time.time()
a = np.arange(l)
b = np.arange(l)
c = a * b
t = time.time() - start
print("Duration: %s" % t)
```

Duration: 0.73 s
Creating NumPy arrays

NumPy arrays can be created from Python structures or by using specific array creation functions.

```python
>>> import numpy as np
>>> a = np.array([1.5, 2.2, 3.0, 0.9])
array([ 1.5, 2.2, 3. , 0.9])
>>> zeros = np.zeros(6)
array([ 0., 0., 0., 0., 0., 0.])
>>> ones = np.ones(6)
array([ 1., 1., 1., 1., 1., 1.])
>>> a = np.arange(12)
>>> print a
[ 0 1 2 3 4 5 6 7 8 9 10 11]
>>> print a.size, a.ndim, a.shape
12 1 (12,)  
>>> m = a.reshape(3, 4)
>>> print m
[[ 0 1 2 3]
 [ 4 5 6 7]
 [ 8 9 10 11]]
>>> print m.size, m.ndim, m.shape
12 2 (3, 4)
>>> Z = zeros((2,3))
>>> print Z
[[ 0. 0. 0.]
 [ 0. 0. 0.]]
>>> v = np.linspace(0, 1.0, 5)
>>> v
array([ 0. , 0.25, 0.5 , 0.75, 1. ])  
```
Array Creation Functions

- `np.array(seq, dtype)`: Creates an array from `seq` having data type `dtype` (optional)
- `np.ones(shape, dtype), np.zeros(shape, dtype)`: Creates an array of given shape and type, filled with ones/zeros. Default type is `float64`.
- `np.arange([start,] stop[, step], dtype)`: Like the normal `range` function but works also with floats. Returns evenly spaced values within a given interval.
- `np.linspace(start, stop[, num])`: Returns evenly spaced numbers over a specified interval.

**`arange` vs. `linspace`:**
- `np.arange(0.0, 1.0, 0.25) ⇒ [0.0, 0.25, 0.5 0.75]
- `np.linspace(0.0, 1.0, 5) ⇒ [0.0, 0.25, 0.5, 0.75, 1.0]`
Indexing Arrays

Python Interpreter

```python
>>> import numpy as np
>>> a = np.arange(20)
>>> a = a.reshape(5,4)
>>> a[3,2]
14
```

5×4 matrix

\[
\begin{pmatrix}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 \\
16 & 17 & 18 & 19 \\
\end{pmatrix}
\]
Indexing Arrays

Python Interpreter

```python
>>> import numpy as np
>>> a = np.arange(20)
>>> a = a.reshape(5,4)
>>> a[3,2]
14
>>> a[1]  # second row
array([4, 5, 6, 7])
```

5 × 4 matrix

$$
\begin{pmatrix}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 \\
16 & 17 & 18 & 19 \\
\end{pmatrix}
$$
Indexing Arrays

Python Interpreter

```python
>>> import numpy as np
>>> a = np.arange(20)
>>> a = a.reshape(5,4)
>>> a[3,2]
14
>>> a[1]  # second row
array([4, 5, 6, 7])
>>> a[-2]  # second last row
array([12, 13, 14, 15])
```

5 × 4 matrix

$$\begin{pmatrix}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 \\
16 & 17 & 18 & 19 \\
\end{pmatrix}$$
Indexing Arrays

**Python Interpreter**

```python
>>> import numpy as np
>>> a = np.arange(20)
>>> a = a.reshape(5,4)
>>> a[3,2]
14
>>> a[1]          # second row
array([4, 5, 6, 7])
>>> a[-2]        # second last row
array([12, 13, 14, 15])
>>> a[:,0]       # first column
array([ 0, 4, 8, 12, 16])
```

5 × 4 matrix

\[
\begin{pmatrix}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 \\
16 & 17 & 18 & 19
\end{pmatrix}
\]
Indexing Arrays

Python Interpreter

```python
>>> import numpy as np
>>> a = np.arange(20)
>>> a = a.reshape(5,4)
>>> a[3,2]
14
>>> a[1]  # second row
array([4, 5, 6, 7])
>>> a[-2]  # second last row
array([12, 13, 14, 15])
>>> a[:,0]  # first column
array([0, 4, 8, 12, 16])
>>> a[1:4, 0:3]  # sub-array
array([[4, 5, 6],
       [8, 9, 10],
       [12, 13, 14]])
```

```
5 \times 4 matrix

\begin{pmatrix}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 \\
16 & 17 & 18 & 19 \\
\end{pmatrix}
```
Indexing Arrays

Python Interpreter

```python
>>> import numpy as np
>>> a = np.arange(20)
>>> a = a.reshape(5,4)
>>> a[3,2]
14
>>> a[1]  # second row
array([4, 5, 6, 7])
>>> a[-2]  # second last row
array([12, 13, 14, 15])
>>> a[0]  # first column
array([0, 4, 8, 12, 16])
>>> a[1:4, 0:3]  # sub-array
array([[4, 5, 6],
       [8, 9, 10],
       [12, 13, 14]])
>>> a[::2, ::3]  # skipping indices
array([[0, 3],
       [8, 11],
       [16, 19]])
```

5 × 4 matrix

```
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
```
Functions on numpy arrays

- The worst thing you can do is iterating with a `for`-loop over a numpy array.
- That’s why numpy supports several standard functions on arrays.

**Python Interpreter**

```python
>>> import numpy as np
>>> a = np.arange(1, 21)
>>> a
[1, 2, 3, 4, ..., 20]
>>> print a.min(), a.max(), a.mean()  # minimum, maximum, arithmetic mean
1 20 10.5
>>> print a.std(), a.var()  # standard deviation, variance
5.76 33.25
>>> print a.sum(), a.prod()  # sum, product
210 2432902008176640000
>>> print a.any(), a.all()  # any True?, all True?
True True
>>> b = np.array([0, 0, 1])
>>> print b.any(), b.all()  # any True?, all True?
True False
```
Arithmetic operations on arrays

- NumPy supports arithmetic operations between arrays
- Advantage: No for-loops necessary (looping occurs in C)
- Element-wise operation for arrays of the same shape

Python Interpreter

```python
>>> import numpy as np
>>> a, b = np.arange(1, 11), np.arange(1,11)
>>> a
array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> a + 1
array([ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
>>> a * 2
array([ 2, 4, 6, 8, 10, 12, 14, 16, 18, 20])
>>> a + b
array([ 2, 4, 6, 8, 10, 12, 14, 16, 18, 20])
>>> a * b
array([ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100])
```

Things get little more complicated when arrays have different shapes. (see Broadcasting)
Operations on arrays of different shapes

- **broadcasting** describes how numpy treats arrays of different shapes during arithmetic operations
- Two dimensions are compatible when they are equal or one of the dimensions is 1

**Python Interpreter**

```python
>>> import numpy as np
>>> a = np.arange(9.0)
>>> a = a.reshape((3,3))
>>> a
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a * b
array([[ 0.,  2.,  6.],
       [ 3.,  8., 15.],
       [ 6., 14., 24.]])
```

a (2d array): 3 x 3  
b (1d array): x 3  
Result : 3 x 3  

The smaller array gets broadcasted to the larger array. Thus, result is computed by element-wise multiplication of

\[
\begin{pmatrix}
0 & 1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8 \\
\end{pmatrix}
\cdot
\begin{pmatrix}
1 & 2 & 3 \\
1 & 2 & 3 \\
1 & 2 & 3 \\
\end{pmatrix}
\]
Matrices are special array objects
Always 2-dimensional
Matrix multiplication
special properties of matrices:
  - matrix.I (Inverse)
  - matrix.T (Transposed)
  - matrix.H (Conjugate)
  - matrix.A (Array conversion)

Python Interpreter

```python
>>> import numpy as np
>>> m = np.matrix([[1, 2], [3,4]])
>>> m
matrix([[1, 2],
        [3, 4]])
>>> m.I
matrix([[[-2. , 1. ],
          [ 1.5, -0.5]]])
>>> m.T
matrix([[1, 3],
        [2, 4]])
>>> b = np.array([2, 3])
>>> b.shape = (2, 1)
>>> b
array([[2],
       [3]])
>>> m * b
matrix([[ 8],
        [18]])
```

Felix Steffenhagen (Uni Freiburg)
The `numpy.linalg` submodule provides core linear algebra tools.

```python
>>> import numpy as np
>>> import numpy.linalg as linalg

A = np.matrix([[2, 3, -1],
                [1, 3, 1],
                [-2, -2, 4]])

>>> A
matrix([[ 2, 3, -1],
        [ 1, 3, 1],
        [-2, -2, 4]])

>>> b = np.array([1, 2, 4])

>>> linalg.solve(A, b)
array([ 3., -1.,  2.])
```

\[
\begin{align*}
2x + 3y - z &= 1 \\
x + 3y + z &= 2 \\
-2x - 2y + 4z &= 4
\end{align*}
\]
Polynoms

- NumPy defines a polynom datatype that allows symbolic computations value evaluation and polynomial arithmetic
- Derivation, Integration

```python
p = np.poly1d(coefs)
```

- Constructs a polynom \( p \) from the given coefficient sequence ordered in decreasing power.

```python
p.deriv(m), p.integ(m)
```

- Compute the derivative or anti-derivative of \( p \). Parameter \( m \) determines the order of derivation.

```
e.g. \( f(x) = 3x^2 - 2x + 1 \)
```

```python
>>> import numpy as np
>>> f = np.poly1d([3, -2, 1])
>>> print(f)
2
3 x - 2 x + 1
>>> f(2.5)
14.75
>>> f_1, F = f.deriv(), f.integ()
>>> print f_1
6 x - 2
>>> print F
3 2
1 x - 1 x + 1 x
```
Curve fitting

- polynomial regression
- \texttt{np.polyfit(x, y, deg)}:
  Least squares polynomial fit of degree \texttt{deg} for coordinate sequences \texttt{x} and \texttt{y}.
  Returns array with polynomial coefficients.

Python Interpreter

```python
from numpy import array, poly1d, polyfit
>>> x = array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
>>> y = array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
```

![Plot showing polynomial fitting](image)
Curve fitting

- polynomial regression
- `np.polyfit(x, y, deg)`:
  Least squares polynomial fit of degree `deg` for coordinate sequences `x` and `y`. Returns array with polynomial coefficients.

Python Interpreter

```python
from numpy import array, poly1d, polyfit
>>> x = array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
>>> y = array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
# linear fit
>>> coefs = polyfit(x, y, 1)
>>> p1 = poly1d(coefs)
>>> print(p1)
-0.3029 x + 0.7571
```

![Graph showing linear fit with points and line](image)
Curve fitting

- polynomial regression
- `np.polyfit(x, y, deg)`: Least squares polynomial fit of degree `deg` for coordinate sequences `x` and `y`. Returns array with polynomial coefficients.

Python Interpreter

```python
from numpy import array, poly1d, polyfit
>>> x = array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
>>> y = array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
# linear fit
>>> coefs = polyfit(x, y, 1)
>>> p1 = poly1d(coefs)
>>> print(p1)
-0.3029 x + 0.7571
# cubical fit
>>> coefs = polyfit(x, y, 3)
>>> p3 = poly1d(coefs)
>>> print(p3)
3      2
0.08704 x - 0.8135 x + 1.693 x - 0.03968
```
SciPy package

- Collection of mathematical algorithms and convenience functions
- Built on NumPy
- Organized into sub-packages

Some of the interesting modules:

<table>
<thead>
<tr>
<th>Sub-Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cluster</td>
<td>Clustering algorithms</td>
</tr>
<tr>
<td>constants</td>
<td>Physical Constants</td>
</tr>
<tr>
<td>fftpack</td>
<td>Fast Fourier Transformation</td>
</tr>
<tr>
<td>integrate</td>
<td>Integration and ODE solvers</td>
</tr>
<tr>
<td>interpolate</td>
<td>Interpolation (e.g. Splines)</td>
</tr>
<tr>
<td>special</td>
<td>Special functions</td>
</tr>
<tr>
<td></td>
<td>(e.g. Bessel functions, Gamma-Function)</td>
</tr>
<tr>
<td>stats</td>
<td>Statistical Functions and Distributions</td>
</tr>
</tbody>
</table>

SciPy-Safari: Integration

```python
import scipy.integrate as spint

def f(x):
    return x**2

spint.quad(f, 0, 2)
output: (2.66..., 2.96e-14)
```

SciPy also supports infinite integration limits. See documentation.
import numpy as np
import scipy.interpolate as spintp

x = np.linspace(0, 2 * np.pi, 10)
y = np.sin(x)

x_spline = np.linspace(0, 2 * np.pi, 100)
y_spline = spintp.spline(x, y, x_spline)
y_spline

# output:
array([ 3.851e-16, 6.465e-02, 1.286e-01,
       1.917e-01, 2.538e-01, 3.146e-01,
       3.739e-01, 4.317e-01, 4.876e-01,
       5.416e-01, 5.934e-01, 6.427e-01,
       6.896e-01, 7.337e-01, 7.749e-01,
       8.132e-01, 8.483e-01, 8.801e-01,
       9.085e-01, 9.333e-01, 9.544e-01,
       ...])
Random Number Generation

numpy.random sub module (= scipy.random) provides many different functions for random number generation.

- sp.rand(d0, d1, ...):
  Create array of given shape filled with uniform random numbers over [0, 1].

- sp.randn(d0, d1, ...):
  The same as sp.rand() but generates zero-mean unit-variance Gaussian random numbers.

- sp.random.randint(low, high=None, size=None):
  Return random integers x such that low \leq x < high. If high is None, then 0 \leq x < low.

- sp.random.binomial(n, p, shape=None):
  Draw n samples from binomial distr. with success probability p. Returns array of given shape containing the number of successes.
Random Number Generation - Examples

Python Interpreter

```python
>>> from scipy.random import *  # import all random functions
>>> rand(2,3)  # 2x3 array
array([[ 0.49010722,  0.73308678,  0.5209828 ],
       [ 0.54217486,  0.75698016,  0.10697513]])
>>> rnd = randn(100)  # 100 norm. distr. numbers
>>> rnd.mean()  # mean should be close to 0
0.0789
>>> randint(1, 50, 6)  # lottery numbers 6 of 49
array([ 2, 28, 15, 49, 22, 35])
>>> binomial(5, 0.4)  # unfair coin flipping
2
>>> binomial(5, 0.4, 10)  # 10 games with 5 flips
array([4, 3, 0, 1, 3, 2, 3, 2, 1, 3])
```
Data Visualization with matplotlib

- **matplotlib** provides 2D data visualization as in MATLAB.
  - Publication quality plots
  - Export to different file formats
  - Embeddable in graphical user interfaces
  - Making plots should be easy!
- Heavy use of NumPy and SciPy
- **pylab**: provides a matlab-like environment
  (roughly: combines NumPy, SciPy and matplotlib)

<table>
<thead>
<tr>
<th><strong>pylab (matplotlib.pyplot)</strong></th>
<th>(Provides plot functions similar to MATLAB)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>matplotlib API</strong></td>
<td>(Basic libraries for creating and managing figures, text, lines, ... )</td>
</tr>
<tr>
<td><strong>Backend</strong></td>
<td>(device dependent renderers)</td>
</tr>
</tbody>
</table>
A simple plot

Plots are generated successively. Each plotting function makes changes to the figure.

Python Interpreter

```python
>>> from pylab import *
# Turn on interactive mode
>>> ion()
# 10 norm. distr. rnd numbers
>>> x = randn(10)
>>> plot(x)
```
A simple plot

Plots are generated successively. Each plotting function makes changes to the figure.

Python Interpreter

```python
from pylab import *

# Turn on interactive mode
ion()

# 10 norm. distr. rnd numbers
x = randn(10)

# setting axis limits
axis([0, 10, -3, 3])
```
A simple plot

Plots are generated successively. Each plotting function makes changes to the figure.

Python Interpreter

```python
>>> from pylab import *
# Turn on interactive mode
>>> ion()
# 10 norm. distr. rnd numbers
>>> x = randn(10)
>>> plot(x)
# setting axis limits
>>> axis([0, 10, -3, 3])
# toggle grid
>>> grid()
```
A simple plot

Plots are generated successively. Each plotting function makes changes to the figure.

Python Interpreter

```python
>>> from pylab import *
# Turn on interactive mode
>>> ion()
# 10 norm. distr. rnd numbers
>>> x = randn(10)
>>> plot(x)
# setting axis limits
>>> axis([0, 10, -3, 3])
# toggle grid
>>> grid()
>>> grid()
```
A simple plot

Plots are generated successively. Each plotting function makes changes to the figure.

Python Interpreter

```python
>>> from pylab import *
# Turn on interactive mode
>>> ion()
# 10 norm. distr. rnd numbers
>>> x = randn(10)
>>> plot(x)
# setting axis limits
>>> axis([0, 10, -3, 3])
# toggle grid
>>> grid()
>>> grid()
# add another plot
>>> y = linspace(-3, 3, 10)
>>> plot(y)
```
A simple plot

Plots are generated successively. Each plotting function makes changes to the figure.

Python Interpreter

```python
>>> from pylab import *
# Turn on interactive mode
>>> ion()
# 10 norm. distr. rnd numbers
>>> x = randn(10)
>>> plot(x)
# setting axis limits
>>> axis([0, 10, -3, 3])
# toggle grid
>>> grid()
>>> grid()
# add another plot
>>> y = linspace(-3, 3, 10)
>>> plot(y)
# plot with x and y axis values
>>> x = linspace(0, 9, 100)
>>> plot(x, sin(x))
```
Basic Plotting Functions

- **plot([x,] y):**
  Generates simple line plot for x and y values. If x-values are not specified the array index values (0, 1, 2, ...) will be used.

- **axis(v):**
  Sets the axis limits to the values $v = [\text{xmin}, \text{xmax}, \text{ymin}, \text{ymax}]$. $v$ can also be a string (e.g. 'off', 'equal', 'auto')

- **xlabel(s), ylabel(s):**
  Set labels for x or y axis to s.

- **title(s), suptitle(s):**
  Set title for current plot or for the whole figure.

- **show():**
  Shows the current figure. Usually the last function to be called in a script after generating a plot.

- **clf():** clear the figure
The `plot` function accepts a pattern string specifying the line and symbol style in the format: "<color><line><symbol>"

**example**

```python
# initialize some values
>>> values = arange(10)
# plot red dotted line with circles
>>> plot(x, "r:o")
# plot green dashed line
>>> plot(x + 5, "g--")
```

<table>
<thead>
<tr>
<th>Line Colors</th>
<th>Line Styles</th>
<th>Marker Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>r red</td>
<td>solid line</td>
<td>o circles</td>
</tr>
<tr>
<td>g green</td>
<td>dashed line</td>
<td>s squares</td>
</tr>
<tr>
<td>b blue</td>
<td>dash-dot line</td>
<td>x crosses</td>
</tr>
<tr>
<td>w white</td>
<td>dotted line</td>
<td>+ plus’es</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* stars</td>
</tr>
<tr>
<td>c cyan</td>
<td></td>
<td>D diamonds</td>
</tr>
<tr>
<td>m magenta</td>
<td></td>
<td>d thin diamonds</td>
</tr>
<tr>
<td>y yellow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k black</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Let’s make a plot having labels, title and a legend.

Python Interpreter

```python
>>> x = linspace(-5, 5, 100)
>>> y_sin, y_cos = sin(x), cos(x)
```

This is the plot we want to create.
Adding Labels and Legends

Let's make a plot having labels, title and a legend.

Python Interpreter

```python
>>> x = linspace(-5, 5, 100)
>>> y_sin, y_cos = sin(x), cos(x)
>>> plot(x, y_sin, "r", label="sine")
```

Adding the sine curve.
Let’s make a plot having labels, title and a legend.

```
>>> x = linspace(-5, 5, 100)
>>> y_sin, y_cos = sin(x), cos(x)
>>> plot(x, y_sin, "r", label="sine")
>>> plot(x, y_cos, "b", label="cosine")
```

Adding the cosine curve.
Let's make a plot having labels, title and a legend.

Python Interpreter

```python
>>> x = linspace(-5, 5, 100)
>>> y_sin, y_cos = sin(x), cos(x)
>>> plot(x, y_sin, "r", label="sine")
>>> plot(x, y_cos, "b", label="cosine")
>>> xlabel("x-value")
>>> ylabel("y-value")
>>> title("Sine and Cosine function")
```

Adding labels and title.
Let's make a plot having labels, title and a legend.

Python Interpreter

```python
>>> x = linspace(-5, 5, 100)
>>> y_sin, y_cos = sin(x), cos(x)
>>> plot(x, y_sin, "r", label="sine")
>>> plot(x, y_cos, "b", label="cosine")
>>> xlabel("x-value")
>>> ylabel("y-value")
>>> title("Sine and Cosine function")
>>> axis([-5, 5, -2, 2])
>>> grid()
```

Changing axis and adding grid.
Let’s make a plot having labels, title and a legend.

```python
>>> x = linspace(-5, 5, 100)
>>> y_sin, y_cos = sin(x), cos(x)
>>> plot(x, y_sin, "r", label="sine")
>>> plot(x, y_cos, "b", label="cosine")
>>> xlabel("x-value")
>>> ylabel("y-value")
>>> title("Sine and Cosine function")
>>> axis([-5, 5, -2, 2])
>>> grid()
>>> legend(loc="upper center")
```

Add the legend.
Let’s make a plot having labels, title and a legend.

Python Interpreter

```python
>>> x = linspace(-5, 5, 100)
>>> y_sin, y_cos = sin(x), cos(x)
>>> plot(x, y_sin, "r", label="sine")
>>> plot(x, y_cos, "b", label="cosine")
>>> xlabel("x-value")
>>> ylabel("y-value")
>>> title("Sine and Cosine function")
>>> axis([-5, 5, -2, 2])
>>> grid()
>>> legend(loc="upper center")
>>> annotate(r"\pi" around here,
        xy=(3.1, 1.0), xytext=(-1.0, -1.5),
        arrowprops=dict(color='black'))
```

Add the annotation.
The complete plot

Sine and Cosine Function

\[ y \approx \pi \text{ around here} \]

\[ \sin x \text{ and } \cos x \]
Annotations and Text functions

- **legend()**: Adds a legend to the current plot. Use keyword parameter `loc` to set the location either by string (e.g. `'upper center'`) or by 2-tuple (e.g. `(2, 3)`).

- **annotate(text, xy=(ax, ay), xytext=(tx, ty))**: Annotate special location `(ax, ay)` and put text at location `(tx, ty)`.
  - Optional parameter `arrowprops` is a dictionary of arrow properties. If properties are set, an arrow is drawn in the figure.

- **text(x, y, text)**: Add text at location `(x, y)`.

Wherever text can be added (labels, titles, annotations), you can use TeX formulas (e.g. `r"$\sum_i^n i$"`). `r" "` is a raw string in which backslashes are kept unchanged.
- `matplotlib` provides a lot of plot types
- Lineplots, Scatterplots, Histograms, Timeseries plots, ...

http://matplotlib.sourceforge.net/gallery.html
hist(x, bins=10)

Computes and draws the histogram of x. Additional keyword options:

- normed=[False | True]: normalize to probability density
- orientation=["horizontal" | "vertical"]

Python Interpreter

```python
# create some data
>>> mu, sigma = 3, 1.2
>>> values = mu + sigma * randn(100)
# plot histogram
>>> hist(values, normed=True,
       color="#42da42", ec="black")
```
### Histograms

- `hist(x, bins=10)`
  Computes and draws the histogram of `x`. Additional keyword options:
  - `normed=[False | True]`: normalize to probability density
  - `orientation=["horizontal" | "vertical"]`

#### Python Interpreter

```python
# create some data
>>> mu, sigma = 3, 1.2
>>> values = mu + sigma * randn(100)
# plot histogram
>>> hist(values, normed=True,
       color="#42da42", ec="black")

# add Norm PDF
>>> p = gca()
>>> x_min, x_max = p.get_xlim()
>>> x = linspace(x_min, x_max, 100)
>>> plot(x, normpdf(x, mu, sigma))
```

`gca()`: get current axes
- `bar(left, height)`: Make a bar plot with rectangles.
- `xticks(pos, labels)`: Set locations and labels of the xticks

**Python Interpreter**

```python
>>> left = [1, 2, 3]
>>> height = [5, 10, 20]
>>> bar(left, height)
```
Bar Plots

- `bar(left, height)`: Make a bar plot with rectangles.
- `xticks(pos, labels)`: Set locations and labels of the xticks.

```python
>>> left = [1, 2, 3]
>>> height = [5, 10, 20]
>>> bar(left, height)
>>> clf()
>>> bar(left, height, align="center")
```
Bar Plots

- `bar(left, height)`: Make a bar plot with rectangles.
- `xticks(pos, labels)`: Set locations and labels of the xticks.

```
>>> left = [1, 2, 3]
>>> height = [5, 10, 20]
>>> bar(left, height)
>>> clf()
>>> bar(left, height, align="center")
>>> xticks(left, ("A", "B", "C"))
```

![Bar Plot Example](image.png)
Bar Plots for different groups require the separate plotting of each group.

Python Interpreter

```python
>>> bar_width = .5
>>> group1 = [20, 25, 18, 29]
>>> group2 = [22, 24, 25, 35]
```
Bar Plots for different groups require the separate plotting of each group.

Python Interpreter

```python
>>> bar_width = .5
>>> group1 = [20, 25, 18, 29]
>>> group2 = [22, 24, 25, 35]

>>> pos1 = arange(4) + 1
>>> bar(pos1, group1, color="blue")
```
Bar Plots for different groups require the separate plotting of each group.

Python Interpreter

```python
>>> bar_width = .5
>>> group1 = [20, 25, 18, 29]
>>> group2 = [22, 24, 25, 35]

>>> pos1 = arange(4) + 1
>>> bar(pos1, group1, color="blue")

>>> pos2 = pos1 + bar_width + .1
>>> bar(pos2, group2, color="yellow")
```
Bar Plots for different groups require the separate plotting of each group.

```
>>> bar_width = .5
>>> group1 = [20, 25, 18, 29]
>>> group2 = [22, 24, 25, 35]

>>> pos1 = arange(4) + 1
>>> bar(pos1, group1, color="blue")

>>> pos2 = pos1 + bar_width + .1
>>> bar(pos2, group2, color="yellow")

>>> cond = ('C 1', 'C 2', 'C 3', 'C 4')
>>> xticks(pos2, cond)
```
Plotting 2D Arrays (as Images)

- `imshow(X[, cmap])`: Display the image or float array in X. The parameter `cmap` lets you specify a colormap (e.g. `cmap=cm.gray`)
- `colorbar()`: adds a colorbar to the current plot

Python Interpreter

```python
>>> img_dat = rand(30,30)
>>> imshow(img_dat)
```

(see help(colormaps) for more themes)
Plotting 2D Arrays (as Images)

- `imshow(X[, cmap])`: Display the image or float array in `X`. The parameter `cmap` lets you specify a colormap (e.g. `cmap=cm.gray`)
- `colorbar()`: adds a colorbar to the current plot

Python Interpreter

```python
>>> img_dat = rand(30,30)
>>> imshow(img_dat)
>>> colorbar()
```

(see `help(colormaps)` for more themes)
Plotting 2D Arrays (as Images)

- `imshow(X[, cmap])`: Display the image or float array in \( X \). The parameter `cmap` lets you specify a colormap (e.g. `cmap=cm.gray`)
- `colorbar()`: adds a colorbar to the current plot

Python Interpreter

```python
>>> img_dat = rand(30,30)
>>> imshow(img_dat)
>>> colorbar()
>>> gray()
```

(see help(colormaps) for more themes)
Plotting 2D Arrays (as Images)

- `imshow(X[, cmap])`:
  Display the image or float array in X. The parameter `cmap` lets you specify a colormap (e.g. `cmap=cm.gray`)
- `colorbar()`: adds a colorbar to the current plot

**Python Interpreter**

```python
>>> img_dat = rand(30,30)
>>> imshow(img_dat)
>>> colorbar()
>>> gray()
>>> copper()
```

(see `help(colormaps)` for more themes)
Multiple figures and subplots

- matplotlib uses concept of current figures and current plots.
- plot command changes current subplot in current figure.
- arbitrary number of figures and subplots possible
- Plots are arranged in a matrix grid.

![4 in 1 plots]
Multiple figures and subplots cntd.

- Let's create two figures, with two plots in each.
- One aligned horizontally, the other vertically

Python Interpreter

```python
# get some data
>>> x = randn(100)
# create 1st figure
>>> figure(1)
>>> subplot(2,1,1)
>>> hist(x)
>>> subplot(2,1,2)
>>> plot(x)
```

`subplot(rows, cols, n)` creates or switches to n-th plot in a rows×cols arrangement
Let’s create two figures, with two plots in each. One aligned horizontally, the other vertically.

Python Interpreter

```python
# get some data
>>> x = randn(100)
# create 1st figure
>>> figure(1)
>>> subplot(2,1,1)
>>> hist(x)
>>> subplot(2,1,2)
>>> plot(x)

# create 2nd figure
>>> figure(2)
>>> subplot(1,2,1)
>>> hist(x)
>>> subplot(1,2,2)
>>> plot(x)
```
More complex layouts

- `subplot` command allows creation of more complex plot arrangements
- limited to matrix arrangement, no spanning over several cols/rows

- Plot 1 is first plot in $2 \times 2$ layout
- Plot 2 is second plot in $2 \times 2$ layout
- Plot 3 is second plot in $2 \times 1$ layout
Example

Python Interpreter

```python
# generate some data
>>> x, y = randn(100), randn(100)
# generate 1st subplot
>>> subplot(2,2,1)
>>> hist(x)
```
Example

**Python Interpreter**

```python
# generate some data
>>> x, y = randn(100), randn(100)
# generate 1st subplot
>>> subplot(2,2,1)
>>> hist(x)
# generate 2nd subplot
>>> subplot(2,2,2)
>>> plot(x, y, "bo")
```

![Graphs showing histogram and scatter plot]
Example

Python Interpreter

# generate some data
>>> x, y = randn(100), randn(100)
# generate 1st subplot
>>> subplot(2,2,1)
>>> hist(x)
# generate 2nd subplot
>>> subplot(2,2,2)
>>> plot(x, y, "bo")
# generate 3rd subplot
>>> subplot(2,1,2)
>>> plot(x)
Example

Python Interpreter

```python
# generate some data
>>> x, y = randn(100), randn(100)
# generate 1st subplot
>>> subplot(2,2,1)
>>> hist(x)
# generate 2nd subplot
>>> subplot(2,2,2)
>>> plot(x, y, "bo")
# generate 3rd subplot
>>> subplot(2,1,2)
>>> plot(x)
# switch back to plots
>>> subplot(2,2,1)
>>> title("Histogram")
>>> subplot(2,2,2)
>>> title("Scatter Plot")
```
Figures can be saved from interactive window or with function `savefig`.

```python
savefig(filename):
    Saves the current figure as PNG to `filename`.
    Optional keyword parameters:
    - `format`: 'png', 'pdf', 'ps', 'eps', 'svg'
    - `transparent`: If `True` makes the figure transparent
```

```python
from pylab import *
x = linspace(-3, 3, 100)
y = sin(x)
plot(x, y)
savefig("sineplot", format="pdf")
```