# Python in ambiente scientifico



## Introduzione

Python nasce come strumento di calcolo parallelo e distribuito

Numpy
- Array e matrici multi-dimensionali, tensori

Scipy
- Algoritmi, modelli, statistica, integrazione, filtraggio, algebra lineare, ottimizzazione

Matplotlib
- Funzionalità di plotting

# Pacchetti software

Questi strumenti sono presenti nelle principali distribuzioni GNU/Linux sotto forma di pacchetti software

In sistemi Debian-like

- Numpy
  - sudo apt-get install python-numpy
- Scipy
  - sudo apt-get install python-scipy
- Matplotlib
  - sudo apt-get install python-matplotlib



# Numpy

# Il modulo Numpy

Il metodo universalmente accettato di importare il pacchetto numpy è il seguente

```
import numpy as np
```

Motivazioni

- import numpy rende eccessivamente lunghi i riferimenti ai metodi
- from numpy import * rende possibili alcuni clash sui nomi



# Array

L'oggetto più importante del pacchetto numpy è indubbiamente l'array

Un array è simile ad una lista

Differenze con la lista:

- Omogeneità: tutti gli elementi dell'array devono essere dello stesso tipo (tipicamente numerico, ad esempio int o float)
- Efficienza: gli array sono progettati per essere molto efficienti sulle grandi dimensioni

# Creazione array

- Un array viene creato tramite il metodo costruttore array()
- Due argomenti
  - Una lista contenente i valori
  - Una specifica del tipo di dato

```
>>> a = np.array([1, 4, 5, 8], float)
>>> a
array([ 1., 4., 5., 8.])
>>> type(a)
<type 'numpy.ndarray'>
```



# Manipolazione array

- La manipolazione di un array è identica a quella vista per le liste
- Slicing

```
>>> a[:2]
Array([ 1., 4.])
```

- Accesso

```
>>> a[3]
8.0
```

- Modifica

```
>>> a[0] = 5.
>>> a
array([ 5., 4., 5., 8.])
```

# Array multidimensionali

- Gli array possono essere multidimensionali
- Si forniscono molteplici liste di valori
- Costruzione di una matrice

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1., 2., 3.],
       [ 4., 5., 6.]])
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a[1,:]
array([ 4., 5., 6.])
>>> a[:,2]
array([ 3., 6.])
>>> a[-1:,-2:]
array([[ 5., 6.]])
```



# Inizializzazione

- Gli array possono essere inizializzati in diversi modi
- Uso del metodo range()

```
>>>a = np.array(range(6), float).reshape((2, 3))
>>> a
array([[ 0., 1., 2.],
       [ 3., 4., 5.]])
```

- Uso del metodo arange()

```
>>> np.arange(5, dtype=float)
array([ 0., 1., 2., 3., 4.])
```

# Inizializzazione

Gli array possono essere inizializzati in diversi modi

Uso dei metodi zeros() e ones()

```
np.ones((2,3), dtype=float)
array([[ 1., 1., 1.],
[ 1., 1., 1.]])
>>> np.zeros(7, dtype=int)
array([0, 0, 0, 0, 0, 0, 0])
```



# Inizializzazione

Il metodo identity() crea una matrice identità

```
>>> np.identity(4, dtype=float)
array([[ 1., 0., 0., 0.],
[ 0., 1., 0., 0.],
[ 0., 0., 1., 0.],
[ 0., 0., 0., 1.]])
```

Il metodo eye() crea una matrice identità sulla diagonale k-ma

```
>>> np.eye(4, k=1, dtype=float)
array([[ 0., 1., 0., 0.],
[ 0., 0., 1., 0.],
[ 0., 0., 0., 1.],
[ 0., 0., 0., 0.]])
```

# Proprietà degli array

shape: ritorna una tupla contenente le dimensioni dell'array

```
>>> a.shape
(2, 3)
```

dtype: ritorna il tipo di dato memorizzato nell'array

```
>>> a.dtype
dtype('float64')
```



# Lunghezza e contenuti

len(): ritorna il numero di dimensioni dell'array

```
>>> a.ndim
2
```

value in array: ritorna True se value è nell'array, False altrimenti

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> 2 in a
True
>>> 0 in a
False
```

# Reshaping

Le dimensioni di un array possono essere modificate mediante il metodo reshape()
Nota bene: viene creato un nuovo array

```
>>> a = np.array(range(10), float)
>>> a
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
>>> a = a.reshape((5, 2))
>>> a
array([[ 0., 1.],
[ 2., 3.],
[ 4., 5.],
[ 6., 7.],
[ 8., 9.]])
>>> a.shape
(5, 2)
```



# Copia

Nel caso, è possibile creare una copia esatta di un array tramite il metodo copy()

```
>>> a = np.array([1, 2, 3], float)
>>> b = a
>>> c = a.copy()
>>> a[0] = 0
>>> a
array([0., 2., 3.])
>>> b
array([0., 2., 3.])
>>> c
array([1., 2., 3.])
```

# Ordinamento e clipping

Gli elementi di un array sono ordinabili con il metodo sort()

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> a.sort()
>>> a
Array([-1., 0., 2., 5., 6.])
```

Gli elementi di un array esterni ad uno specifico intervallo possono essere filtrati con il metodo clip()

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> a.clip(0, 5)
array([ 5., 2., 5., 0., 0.])
```



# Generazione di una matrice trasposta

Si usa il metodo transpose() per trasporre un array multidimensionale

```
>>> a = np.array(range(6), float).reshape((2, 3))
>>> a
array([[ 0., 1., 2.],
[ 3., 4., 5.]])
>>> a.transpose()
array([[ 0., 3.],
[ 1., 4.],
[ 2., 5.]])
```

# Trasformazione multi monodimensionale

Dato un array multidimensionale, se ne può costruire la versione monodimensionale tramite il metodo flatten()

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1., 2., 3.],
[ 4., 5., 6.]])
>>> a.flatten()
array([ 1., 2., 3., 4., 5., 6.])
```



# Concatenazione

Il metodo concatenate() permette la concatenazione di due array

La concatenazione avviene, per default, sulle righe (parametro axis=0)

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> b = np.array([[5, 6], [7,8]], float)
>>> np.concatenate((a,b))
array([[ 1., 2.],
[ 3., 4.],
[ 5., 6.],
[ 7., 8.]])
```

La concatenazione può essere anche fatta per colonne (parametro axis=1)

```
>>> np.concatenate((a,b), axis=1)
array([[ 1., 2., 5., 6.],
[ 3., 4., 7., 8.]])
```

# Costanti e simboli standard

Pi greco: np.pi

```
>>> np.pi
3.1415926535897931
```

Costante di eulero: np.e

```
>>> np.e
2.7182818284590451
```

Not a Number (NaN): np.NaN

```
>>> np.NaN
nan
```

Infinito: np.Inf

```
>>> np.Inf
inf
```



# Aritmetica di base

Le operazioni di somma, sottrazione, moltiplicazione, divisione, elevamento a potenza, sono definite con i relativi simboli

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([5,2,6], float)
>>> a + b
array([6., 4., 9.])
>>> a □ b
array([-4., 0., -3.])
>>> a * b
array([5., 4., 18.])
>>> b / a
array([5., 1., 2.])
>>> a % b
array([1., 0., 3.])
>>> b**a
array([5., 4., 216.])
```

# Aritmetica: una avvertenza

Negli array multidimensionali, la moltiplicazione rimane ancora elemento per elemento
- NON È la moltiplicazione matriciale!
- Per quella serve il tipo di dato matrix

```
>>> a = np.array([[1,2], [3,4]], float)
>>> b = np.array([[2,0], [1,3]], float)
>>> a * b
array([[2., 0.], [3., 12.]])
```



# Funzioni standard su array

Numpy definisce una serie di funzioni standard
- abs(), sign(), sqrt(), log(), log10(), exp(), sin(), cos(), tan(), arcsin(), arccos(), arctan(), sinh(), cosh(), tanh(), arcsinh(), arccosh(), arctanh()

applicate su ogni elemento

```
>>> a = np.array([1, 4, 9], float)
>>> np.sqrt(a)
array([ 1., 2., 3.])
```

# Funzioni standard su array

Arrotondamento verso il valore più piccolo: floor()

```
>>> a = np.array([1.1, 1.5, 1.9], float)
>>> np.floor(a)
array([ 1., 1., 1.])
```

Arrotondamento verso il valore più grande: ceil()

```
>>> np.ceil(a)
array([ 2., 2., 2.])
```

Arrotondamento verso il valore più vicino: rint()

```
>>> np.rint(a)
array([ 1., 2., 2.])
```



# Funzioni standard su elementi dell'array

È possibile invocare funzioni standard sull'intero set di elementi di un array

sum(), prod()

```
>>> a = np.array([2, 4, 3], float)
>>> a.sum()
9.0
>>> a.prod()
24.0
>>> np.sum(a)
9.0
>>> np.prod(a)
24.0
```

# Funzioni standard su elementi dell'array

È possibile invocare funzioni standard sull'intero set di elementi di un array

mean(), var(), std(), min(), max()

```
>>> a = np.array([2, 1, 9], float)
>>> a.mean()
3.0
>>> a.var()
12.666666666666666
>>> a.std()
3.559026084010437
>>> a.min()
1.0
>>> a.max()
9.0
```



# Funzioni standard su elementi dell'array

È possibile invocare funzioni standard sull'intero set di elementi di un array

argmin(), argmax()

```
>>> a = np.array([2, 1, 9], float)
>>> a.argmin()
1
>>> a.argmax()
2
```

# Funzioni standard su elementi dell'array

Nel caso di array multidimensionali, è possibile specificare su quale dimensione (axis) si vuole operare, indicandola con il parametro axis

```
>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
>>> a.mean(axis=0)     #opera su colonne
array([ 2., 2.])
>>> a.mean(axis=1)     #opera su righe
array([ 1., 1., 4.])
>>> a.min(axis=1)
array([ 0., -1., 3.])
>>> a.max(axis=0)
array([ 3., 5.])
```



# Iteratori

L'iterazione sugli array monodimensionali avviene analogamente a quanto visto per le liste

```
>>> a = np.array([1, 4, 5], int)
>>> for x in a:
... print x
... <hit return>
1
4
5
```

# Iteratori

Per gli array multidimensionali, l'iterazione procede per righe

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for x in a:
... print x
... <hit return>
[ 1. 2.]
[ 3. 4.]
[ 5. 6.]
>>> for (x, y) in a:
... print x * y
... <hit return>
2.0
12.0
30.0
```



# Operatori di confronto

Gli operatori booleani di confronto sono definiti sugli array di uguale dimensione

Il risultato del confronto è un array di valori booleani

```
>>> a = np.array([1, 3, 0], float)
>>> b = np.array([0, 3, 2], float)
>>> a > b
array([ True, False, False], dtype=bool)
>>> a == b
array([False, True, False], dtype=bool)
>>> a <= b
array([False, True, True], dtype=bool)
>>> a > 2
array([False, True, False], dtype=bool)
```

# Operatori di confronto

Il metodo any() ritorna True se almeno un elemento dell'array booleano è True
Il metodo all() ritorna True se tutti gli elementi dell'array booleano sono True

```
>>> c = np.array([ True, False, False], bool)
>>> any(c)
True
>>> all(c)
False
```



# Operatori di confronto

È possibile applicare confronti composti con i metodi logical_and(), logical_or() e logical_not()

```
>>> a = np.array([1, 3, 0], float)
>>> np.logical_and(a > 0, a < 3)
array([ True, False, False], dtype=bool)
>>> b = np.array([True, False, True], bool)
>>> np.logical_not(b)
array([False, True, False], dtype=bool)
>>> c = np.array([False, True, False], bool)
>>> np.logical_or(b, c)
array([ True, True, True], dtype=bool)
```

# Confronto e sostituzione

Il metodo where(condition, if_true, if_false) applica una condizione a tutti gli elementi di un array

In caso di verità logica, applica lo statement if_true sull'elemento

Altrimenti, applica lo statement if_false sull'elemento

```
>>> a = np.array([1, 3, 0], float)
>>> np.where(a != 0, 1 / a, a)
array([ 1. , 0.33333333, 0. ])
```



# Test di valori speciali

È possibile verificare la presenza nell'array di valori non numerici (metodo isnan()) e di valori finiti (metodo isfinite())

```
>>> a = np.array([1, np.NaN, np.Inf], float)
>>> a
array([ 1., NaN, Inf])
>>> np.isnan(a)
array([False, True, False], dtype=bool)
>>> np.isfinite(a)
array([ True, False, False], dtype=bool)
```

# Selezione avanzata

A differenza delle liste, con gli array è possibile effettuare selezioni più raffinate del semplice slicing degli elementi

1)È possibile usare un array selector, ossia un array di booleani i cui valori a True indicano quali valori dell'array originario selezionare

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> a >= 6
array([[ True, False],
[False, True]], dtype=bool)
>>> a[a >= 6]
array([ 6., 9.])
>>> a[np.logical_and(a > 5, a < 9)]
array([ 6.])
```



# Selezione avanzata

A differenza delle liste, con gli array è possibile effettuare selezioni più raffinate del semplice slicing degli elementi

2) È possibile usare un array di indici

```
>>> a = np.array([2, 4, 6, 8], float)
>>> b = np.array([0, 0, 1, 3, 2, 1], int)
>>> a[b]
array([ 2., 2., 4., 8., 6., 4.])
```

# Selezione avanzata

A differenza delle liste, con gli array è possibile effettuare selezioni più raffinate del semplice slicing degli elementi

2b) Il metodo take() seleziona elementi di array i cui indici sono memorizzati in un array di interi

```
>>> a = np.array([2, 4, 6, 8], float)
>>> b = np.array([0, 0, 1, 3, 2, 1], int)
>>> a.take(b)
array([ 2., 2., 4., 8., 6., 4.])
```



# Selezione avanzata

A differenza delle liste, con gli array è possibile effettuare selezioni più raffinate del semplice slicing degli elementi

3)Per gli array multidimensionali, si passano due array di indici

- Il primo array contiene gli indici di riga
- Il secondo array contiene gli indici di colonna

```
>>> a = np.array([[1, 4], [9, 16]], float)
>>> b = np.array([0, 0, 1, 1, 0], int)
>>> c = np.array([0, 1, 1, 1, 1], int)
>>> a[b,c]
array([ 1., 4., 16., 16., 4.])
```

# Selezione avanzata

A differenza delle liste, con gli array è possibile effettuare selezioni più raffinate del semplice slicing degli elementi

4)Nel caso di array multidimensionali, l'argomento axis specifica la selezione per righe o per colonne

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([0, 0, 1], int)
>>> a.take(b, axis=0)
array([[ 0., 1.],
[ 0., 1.],
[ 2., 3.]])
>>> a.take(b, axis=1)
array([[ 0., 0., 1.],
      [ 2., 2., 3.]])
```



# Manipolazione avanzata

Il metodo opposto a take() è put()

put() prende i valori da un array sorgente e li inserisce nell'array destinazione, in una specifica locazione

```
>>> a = np.array([0, 1, 2, 3, 4, 5], float)
>>> b = np.array([9, 8, 7], float)
>>> a.put([0, 3], b)
>>> a
array([ 9., 1., 2., 8., 4., 5.])
>>> a = np.array([0, 1, 2, 3, 4, 5], float)
>>> a.put([0, 3], 5)
>>> a
array([ 5., 1., 2., 5., 4., 5.])
```

# Prodotto scalare e vettoriale

Il prodotto scalare fra due vettori è ottenibile tramite il metodo dot() (dot product)

```
>>> a = np.array([1, 2, 3], float)
>>> b = np.array([0, 1, 1], float)
>>> np.dot(a, b)
5.0
```

Il prodotto vettoriale fra due vettori è ottenibile tramite il metodo cross() (cross product)

```
>>> a = np.array([1, 4, 0], float)
 >>> b = np.array([2, 2, 1], float)
>>> np.cross(a, b)
array([ 4., -1., -6.])
```



# Algebra lineare: il pacchetto linalg

Il sotto-pacchetto np.linalg fornisce gli strumenti di base per l'algebra lineare

Si può calcolare il determinante di una matrice con il metodo det()

```
>>> a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]], float)
>>> a
array([[ 4., 2., 0.],
[ 9., 3., 7.],
[ 1., 2., 1.]])
>>> np.linalg.det(a)
-53.999999999999993
```

# Algebra lineare: il pacchetto linalg

La funzione eig() ritorna una tupla di due array che contengono rispettivamente gli autovalori e gli autovettori della matrice

```
>>> vals, vecs = np.linalg.eig(a)
>>> vals
array([ 9. , 2.44948974, -2.44948974])
>>> vecs
array([[-0.3538921 , -0.56786837, 0.27843404],
[-0.88473024, 0.44024287, -0.89787873],
[-0.30333608, 0.69549388, 0.34101066]])
```



# Algebra lineare: il pacchetto linalg

La funzione inv() ritorna l'inversa di una matrice

```
>>> b = np.linalg.inv(a)
>>> b
array([[ 0.14814815, 0.07407407, -0.25925926],
[ 0.2037037 , -0.14814815, 0.51851852],
[-0.27777778, 0.11111111, 0.11111111]])
```

# Calcolo polinomiale

Il metodo poly() del pacchetto numpy accetta un array contenente le radici di un polinomio e ritorna un array con i coefficienti del polinomio

```
>>> np.poly([-1, 1, 1, 10])
Array([ 1, -11, 9, 11, -10])
```

Il metodo roots() del pacchetto numpy accetta un array contenente i coefficienti del polinomio e ritorna un array con le radici di un polinomio

```
>>> np.roots([1, 4, -2, 3])
array([-4.57974010+0.j , 0.28987005+0.75566815j,
0.28987005-0.75566815j])
```



# Calcolo polinomiale

Sono disponibili le funzioni di somma, sottrazione, moltiplicazione, divisione fra polinomi

polyadd(), polysub(), polymul(), polydiv()

```
>>> print np.polyadd([1, 1, 1, 1], [1, 1, 1, 1])
[2 2 2 2]
```

Il metodo polyval() valuta un polinomio in un punto

```
>>> np.polyval([1, -2, 0, 2], 4)
34
# 1*4^3+(-2)*4^2+0*4^1+2*4^0
```

# Generazione di numeri casuali

Il sotto-pacchetto np.random mette a disposizione strumenti per la generazione di numeri casuali con seme arbitrario

Impostazione del seme

```
>>> np.random.seed(293423)
```

Generazione di un singolo numero casuale uniforme in [0.0, 1.0)

```
>>> np.random.random()
0.70110427435769551
```

Generazione di un singolo numero casuale intero uniforme in un intervallo [a, b]

```
>>> np.random.randint(5, 10)
9
```



# Generazione di distribuzioni

Il pacchetto numpy è in grado di produrre numeri casuali per tutte le principali distribuzioni statistiche

```
>>> np.random.poisson(6.0)   #lambda
5
>>> np.random.normal(1.5, 4.0)   #media, varianza
0.8363655041094318
>>> np.random.normal()
0.2754871694068293 2
>>> np.random.normal(size=5)   #num di valori
array([-1.67215088, 0.65813053, -0.70150614,
 0.91452499, 0.71440557])
```

# Scipy



# Il pacchetto scipy

Il pacchetto scipy utilizza la funzionalità di numpy per fornire un pacchetto di calcolo scientifico general purpose

```
>>>import scipy
```

Scipy è in realtà una collezione enorme di sotto-pacchetti

```
>>> scipy.info(scipy)
```

# Le estensioni offerte

scipy.constants: costanti matematiche e fisiche
scipy.special: funzioni in uso in fisica matematica (ellittiche, Bessel, ipergeometriche)
scipy.integrate: metodi di integrazione numerica (trapezoidale, Simpson), integrazione di equazioni differenziali
scipy.optimize: metodi di ottimizzazione (minimi quadrati, gradiente, simulated annealing)
scipy.linalg: estensione di numpy.linalg; soluzione di sistemi lineari, calcolo matriciale, decomposizione, fattorizzazione
scipy.sparse: gestione di matrici sparse



# Le estensioni offerte

scipy.interpolate: metodi per l'interpolazione lineare e non di dati
scipy.fftpack: Fast Fourier Transform
scipy.signal: metodi di signal processing (filtraggio, correlazione, convoluzione, smoothing)
scipy.stats: distribuzioni di probabilità continue e discrete, calcolo dei momenti, calcolo cumulative, statistica descrittiva, test

# Un esempio: generazione di numeri casuali

Usiamo il pacchetto scipy.stats per produrre un array di valori distribuito secondo una distribuzione Beta(5, 5)  #alpha,beta

```
import scipy.stats
q = scipy.stats.beta(5, 5)# genera una beta(5,5)
obs = q.rvs(2000)         # produce 2000 osservazioni
```

Stampiamo statistiche sull'insieme

```
print obs.min()
0.0749989919902
print obs.max()
0.919066721448
print obs.std()
0.152290115168
print obs.mean()
0.506227887253
```



# Un esempio: regressione lineare

Usiamo il pacchetto scipy.stats per effettuare una regressione lineare

```
import scipy.stats
x = np.arange(1.0, 11.0, 1.0)
y = np.array([1.0, 1.0, 4.0, 3.0, 6.0, 5.0, 8.0, 10.0, 9.0, 11.0])
gradient, intercept, r_value, p_value, std_err = \
  scipy.stats.linregress(x, y)
```

Gradient: coefficiente angolare

Intercept: intersezione con asse Y

R_value: radice quadrata del coefficiente di correlazione

P_value: test statistico sull'ipotesi nulla "il coefficiente angolare della retta di regressione è zero"

Std_err: errore standard della stima

# Matplotlib



# Il pacchetto matplotlib

Il pacchetto matplotlib è una libreria di disegno orientata agli oggetti, che permette di creare grafici di ogni tipo

Il pacchetto pylab implementa una interfaccia procedurale da linea di comando a matplotlib, in stile Matlab

```
>>>import pylab
```

Una galleria di esempi esaustiva può essere trovata al seguente indirizzo: http://www.scipy.org/Cookbook/Matplotlib

# Creazione di un grafico
# Esempio matplot.py

Si importano i moduli necessari

```
import numpy as np
import scipy
import pylab
```

Si genera un intervallo di valori sull'asse delle x

```
t = np.arange(0.0, 1.0, 0.01)
```

Si genera un intervallo di valori sull'asse delle y

```
s = scipy.sin(2*scipy.pi*t)
```

Si genera il grafico

```
pylab.plot(t, s)
```

Attenzione: si è generato il grafico, non lo si è ancora mostrato!



# Esempio matplot.py

Si imposta un nome all'asse delle x

```
pylab.xlabel('time (s)')
```

Si imposta un nome all'asse delle y

```
pylab.ylabel('Voltage (mV)')
```

Si imposta il titolo del grafico

```
pylab.title('Simple graph')
```

Se lo si vuole, si può impostare una griglia sullo sfondo

```
pylab.grid(True)
```

Se lo si vuole, si può salvare il grafico su file

```
pylab.savefig('simple_plot')
```

Infine, visualizziamo il grafico

```
pylab.show()
```

# Generazione di numeri casuali Esempio histogram.py

Mostriamo in un grafico l'istogramma dei valori generati casualmente

```
import numpy as np
import scipy.stats
import pylab
q = scipy.stats.beta(5, 5)# genera una beta(5,5)
obs = q.rvs(2000)          # produce 2000 osservazioni
pylab.hist(obs, bins=40, normed=True)# istogramma
x = np.arange(0.01, 1.01, 0.01)# asse X
pylab.plot(x, q.pdf(x), 'k-', linewidth=2)# grafico PDF
pylab.show()               # mostra il grafico
```



# Un esempio: regressione lineare

Mostriamo in un grafico la regressione lineare

```
import numpy as np
import scipy.stats
import pylab
def f(x, g, i):
  return g*x + i
x = np.arange(1.0, 11.0, 1.0)
y = np.array([1.0, 1.0, 4.0, 3.0, 6.0, 5.0, 8.0, 10.0, 9.0, 11.0])
gradient, intercept, r_value, p_value, std_err = \
  scipy.stats.linregress(x, y)
pylab.plot(x, y, 'ro')
pylab.plot(x, f(x, gradient, intercept), 'k-', linewidth=2)
pylab.show()
```