THE NUMPY STACK IN PYTHON

NUMPY

$$a \bullet b = a^T b = \sum_{d=1}^{D} a_d b_d$$

$$a \bullet b = |a||b| cos\theta_{ab}$$

$$cos\theta_{ab} = \frac{a^T b}{|a||b|}$$

```
In [27]: a = np.array([1,2])

In [28]: b = np.array([2,1])

In [29]: dot = 0

In [30]: for e, f in zip(a, b):
   ....:     dot += e*f
   ....:

In [31]: dot
Out[31]: 4

In [32]:
```

```
In [27]: a = np.array([1,2])

In [28]: b = np.array([2,1])

In [29]: dot = 0

In [30]: for e, f in zip(a, b):
    ....:     dot += e*f
    ....:

In [31]: dot
Out[31]: 4

In [32]: a*b
Out[32]: array([2, 2])

In [33]: np.sum(a*b)
Out[33]: 4

In [34]: (a*b).sum()
Out[34]: 4

In [35]:
```

```
In [29]: dot = 0

In [30]: for e, f in zip(a, b):
   ....:     dot += e*f
   ....:

In [31]: dot
Out[31]: 4

In [32]: a*b
Out[32]: array([2, 2])

In [33]: np.sum(a*b)
Out[33]: 4

In [34]: (a*b).sum()
Out[34]: 4

In [35]: np.dot(a, b)
Out[35]: 4

In [36]: a.dot(b)
Out[36]: 4
```

```
Out[31]: 4

In [32]: a*b
Out[32]: array([2, 2])

In [33]: np.sum(a*b)
Out[33]: 4

In [34]: (a*b).sum()
Out[34]: 4

In [35]: np.dot(a, b)
Out[35]: 4

In [36]: a.dot(b)
Out[36]: 4

In [37]: b.dot(a)
Out[37]: 4

In [38]: amag = np.sqrt( (a*a).sum() )

In [39]: amag
Out[39]: 2.2360679774997898
```

```
In [41]: amag
Out[41]: 2.2360679774997898

In [42]: cosangle = a.dot(b) / ( np.linalg.norm

In [43]: cosangle
Out[43]: 0.79999999999999982

In [44]: angle = np.arccos(cosangle)

In [45]: angle
```

```
In [37]: b.dot(a)
Out[37]: 4

In [38]: amag = np.sqrt( (a*a).sum() )

In [39]: amag
Out[39]: 2.2360679774997898

In [40]: amag = np.linalg.norm(a)

In [41]: amag
Out[41]: 2.2360679774997898

In [42]: cosangle = a.dot(b) / ( np.linalg.norm(a) * np.linalg.norm(b) )

In [43]: cosangle
Out[43]: 0.79999999999999982

In [44]: angle = np.arccos(cosangle)

In [45]: angle
Out[45]: 0.6435011087932847
```

```python
import numpy as np
from datetime import datetime

a = np.random.randn(100)
b = np.random.randn(100)
T = 100000

def slow_dot_product(a, b):
    result = 0
    for e, f in zip(a, b):
        result += e*f
    return result

t0 = datetime.now()
for t in xrange(T):
    slow_dot_product(a, b)
dt1 = datetime.now() - t0

t0 = datetime.now()
for t in xrange(T):
    a.dot(b)
dt2 = datetime.now() - t0

print "dt1 / dt2:", dt1.total_seconds() / dt2.total_seconds()
```

```
In [47]: M = np.array([ [1,2], [3,4] ])

In [48]: L = [ [1,2], [3,4] ]

In [49]: L[0]
Out[49]: [1, 2]

In [50]: L[0][0]
Out[50]: 1

In [51]: M[0][0]
Out[51]: 1

In [52]: M[0,0]
Out[52]: 1

In [53]: M2 = np.matrix([ [1,2], [3,4] ])

In [54]: M2
Out[54]:
matrix([[1, 2],
        [3, 4]])

In [55]:
```

```
In [55]: A = np.array(M2)

In [56]: A
Out[56]:
array([[1, 2],
       [3, 4]])

In [57]: A.T
Out[57]:
array([[1, 3],
       [2, 4]])
```

```
In [59]: np.array([1,2,3])
Out[59]: array([1, 2, 3])

In [60]: Z = np.zeros(10)

In [61]: Z
Out[61]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [62]: Z = np.zeros((10, 10))

In [63]: Z
Out[63]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

```
In [64]: O = np.ones((10, 10))

In [65]: O
Out[65]:
array([[ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]])
```

```
In [66]: R = np.random.random((10,10))

In [67]: R
Out[67]:
array([[ 0.2621964 ,  0.03294989,  0.4264689 ,  0.05322232,  0.45945556,
         0.08863329,  0.0899541 ,  0.46565893,  0.77619219,  0.84913824],
       [ 0.16706504,  0.73525562,  0.6258681 ,  0.2865714 ,  0.99721985,
         0.69168229,  0.15078506,  0.06358177,  0.06739394,  0.11186605],
       [ 0.70743975,  0.0299889 ,  0.22307198,  0.61944724,  0.89262195,
         0.06490885,  0.7471046 ,  0.74378233,  0.47270786,  0.36987678],
       [ 0.79237646,  0.61982643,  0.84691392,  0.13719082,  0.1335911 ,
         0.94989285,  0.28745693,  0.60270758,  0.92807351,  0.73378454],
       [ 0.9494111 ,  0.73071567,  0.75498156,  0.00390129,  0.09507986,
         0.15361064,  0.71035798,  0.2775515 ,  0.51052561,  0.75790385],
       [ 0.82103405,  0.45427536,  0.66966492,  0.85379898,  0.63264582,
         0.86515971,  0.46884648,  0.87218926,  0.61785016,  0.7794741 ],
       [ 0.23574008,  0.65738627,  0.75198749,  0.1567005 ,  0.93822085,
         0.52143264,  0.71650547,  0.65879922,  0.48932995,  0.9401724 ],
       [ 0.61647496,  0.85881235,  0.65531419,  0.19731385,  0.50805796,
         0.45416833,  0.6308419 ,  0.13692046,  0.32498359,  0.04084511],
       [ 0.01790617,  0.34746793,  0.82116272,  0.67454763,  0.97539446,
         0.33594086,  0.77044552,  0.13066116,  0.12370116,  0.94533962],
       [ 0.47320028,  0.74911315,  0.71213753,  0.03135058,  0.8444124 ,
         0.84941261,  0.44294644,  0.93499864,  0.57597144,  0.46319759]])
```

```
In [69]: G = np.random.randn(10,10)

In [70]: G
Out[70]:
array([[ 1.01921493, -1.11467195, -1.65234527, -0.29628681, -1.37111564,
         1.58207891, -0.19773701,  0.6065109 , -0.04952188, -0.34325721],
       [ 0.23295241, -0.06400185,  0.46633317,  0.11136497,  0.71218272,
         0.08934439,  0.01895695, -0.50505558, -0.66636474, -0.05189705],
       [ 1.70708939, -0.92564671,  0.62789392, -0.30484719,  1.30649489,
         0.16349809,  0.74243239, -0.92299132, -1.28488714, -0.320924  ],
       [ 1.1424565 , -0.62181885, -0.36190247,  0.10793852, -0.69898661,
        -0.48361718,  1.94387311, -0.66678632, -2.18687274, -1.01691781],
       [ 0.51172866,  0.77032004,  1.81275475,  0.47825179,  0.49993343,
        -0.12264262,  0.49810333,  1.53774709,  0.92149976, -0.86747378],
       [-1.88335267,  0.22389425,  0.81081607, -0.76811913, -0.99960316,
         0.03337916, -0.71803662, -0.54168553,  0.51096094,  0.31322329],
       [ 0.94460992, -0.42389632, -1.0461626 ,  0.07824807,  0.80377239,
        -0.5311184 , -1.1296973 ,  1.36530634,  0.51185303, -0.72693972],
       [ 0.1956327 , -0.74314481, -0.4711628 ,  0.57318002,  0.49878321,
         1.38010781,  0.58215828,  1.92366223, -0.48416877,  0.16341422],
       [ 0.13789505, -0.48856808,  0.25208325,  0.56962797,  0.18640829,
        -0.53251459, -1.72961735,  0.10999802, -0.22466329, -1.95642366],
       [-0.02515611,  1.15030707,  0.6113032 , -2.12230498,  0.47880963,
        -0.48471963,  1.20596466,  0.99327398, -1.16437374, -1.21584577]])
```

```
In [71]: G.mean()
Out[71]: -0.012902166701024271

In [72]: G.var()
Out[72]: 0.8351978081914514
```

Matrix Products

- Matrix multiplication
- Requirement: inner dimensions must match
- If we have A of size (2,3) and B of size (3,3)
- We can multiply AB (inner dimension is 3)
- We cannot multiply BA (inner dimension is 3 / 2)

Matrix Products

- Matrix multiplication definition:

$$C(i,j) = \sum_{k=1}^{K} A(i,k)B(k,j)$$

- (i, j)th entry of C is the dot product of row A(i, :) and column B(:, j)
- In Numpy: C = A.dot(B)

Matrix Products

- It's very natural to want to do:

C(i, j) = A(i, j) * B(i, j)

- i.e. element-wise multiplication
- We saw that asterisk (*) does this for vectors
- It also works with 2-D arrays, and >2-D arrays
- Both arrays must be same size
- Odd that there is no well-defined mathematical symbol for it
- Shows up a few times in deep learning

$$Element - wise\ Multiplication: \ \otimes \ \ or \ \odot$$

```
In [74]: A = np.array([[1,2],[3,4]])

In [75]: Ainv = np.linalg.inv(A)

In [76]: Ainv
Out[76]:
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

In [77]: Ainv.dot(A)
Out[77]:
array([[  1.00000000e+00,   0.00000000e+00],
       [  2.22044605e-16,   1.00000000e+00]])

In [78]: A.dot(Ainv)
Out[78]:
array([[  1.00000000e+00,   0.00000000e+00],
       [  8.88178420e-16,   1.00000000e+00]])

In [79]:
```

```
In [79]: np.linalg.det(A)
Out[79]: -2.0000000000000004

In [80]: np.diag(A)
Out[80]: array([1, 4])

In [81]: np.diag([1,2])
Out[81]:
array([[1, 0],
       [0, 2]])
```

Outer Product / Inner Product

- Outer product:

C(i,j) = A(i)B(j)

Ex. $$\Sigma = E\{(x-\mu)(x-\mu)^T\} \approx \frac{1}{N-1}\sum_{n=1}^{N}(x_n-\overline{x})(x_n-\overline{x})^T$$

- Inner product:

C = sum over i { A(i)B(i) }

Same as dot product

```
In [82]: a = np.array([1,2])

In [83]: b = np.array([3,4])

In [84]: np.outer(a, b)
Out[84]:
array([[3, 4],
       [6, 8]])
```

```
In [87]: np.diag(A).sum()
Out[87]: 5

In [88]: np.trace(A)
Out[88]: 5

In [89]: X = np.random.randn(100,3)

In [90]: cov = np.cov(X)

In [91]: cov.shape
Out[91]: (100, 100)

In [92]: cov = np.cov(X.T)

In [93]: cov
Out[93]:
array([[ 0.99614149,  0.02035475,  0.02925818],
       [ 0.02035475,  1.01374315, -0.14305382],
       [ 0.02925818, -0.14305382,  0.9794798 ]])
```

Eigenvalues and Eigenvectors

eigenvalues, eigenvectors = np.eig(C)

OR

eigenvalues, eigenvectors = np.eigh(C)

eigh is for symmetric and Hermitian matrices

Symmetric means $A = A^T$

Hermitian means $A = A^H$

A^H = conjugate transpose of A

```
In [95]: np.linalg.eig(cov)
Out[95]:
(array([ 0.84428276,  1.0042688 ,  1.14081287]),
 array([[-0.22707313,  0.97341917,  0.02988163],
        [ 0.64399103,  0.17310181, -0.74519213],
        [ 0.73055687,  0.14996961,  0.66617998]]))

In [96]: np.linalg.eig(cov)
Out[96]:
(array([ 0.84428276,  1.0042688 ,  1.14081287]),
 array([[-0.22707313,  0.97341917,  0.02988163],
        [ 0.64399103,  0.17310181, -0.74519213],
        [ 0.73055687,  0.14996961,  0.66617998]]))

In [97]: np.linalg.eig(cov)
Out[97]:
(array([ 0.84428276,  1.0042688 ,  1.14081287]),
 array([[-0.22707313,  0.97341917,  0.02988163],
        [ 0.64399103,  0.17310181, -0.74519213],
        [ 0.73055687,  0.14996961,  0.66617998]]))
```

Solving a Linear System

Problem: $Ax = b$

Solution: $A^{-1}Ax = x = A^{-1}b$

- Is a system of D equations and D unknowns
- A is DxD, assume it is invertible
- We have all the tools we need to solve this already:
  - Matrix inverse
  - Matrix multiply (dot)

```
In [99]: A
Out[99]:
array([[1, 2],
       [3, 4]])

In [100]: b
Out[100]: array([3, 4])

In [101]: b = np.array([1,2])

In [102]: b
Out[102]: array([1, 2])

In [103]: x = np.linalg.inv(A).dot(b)

In [104]: x
Out[104]: array([ 0. ,  0.5])

In [105]:
```

Solving a Linear System

- In MATLAB, you get a warning if you try to do inv(A)*b
- In MATLAB it's not called solve(), but it uses the same algorithm
- It is both more efficient and more accurate
- So always use solve(), never use the inverse method

Example Problem

The admission fee at a small fair is $1.50 for children and $4.00 for adults. On a certain day, 2200 people enter the fair and $5050 is collected. How many children and how many adults attended?

Let:

X1 = number of children, X2 = number of adults

X1 + X2 = 2200

1.5X1 + 4X2 = 5050

$$\begin{pmatrix} 1 & 1 \\ 1.5 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2200 \\ 5050 \end{pmatrix}$$

```
In [108]: A = np.array([[1,1], [1.5,4]])

In [109]: b = np.array([2200, 5050])

In [110]: np.linalg.solve(A, b)
Out[110]: array([ 1500.,    700.])

In [111]:
```

PANDAS

```
object?   -> Details about 'object', use 'object??' 

In [1]: X = []

In [2]: import numpy as np

In [3]: for line in open("data_2d.csv"):
   ...:     row = line.split(',')
   ...:     sample = map(float, row)
   ...:     X.append(sample)
   ...:
```

```
 [4.1426693978, 52.2547263792, 168.034400947]]

In [5]: X = np.array(X)

In [6]: X.shape
Out[6]: (100, 3)
```

DataFrames

- Our first look at Pandas
- It works a lot like R (if you come from an R background)
- If you're not familiar with R, some things that Pandas does might seem backwards or contrary to the way Numpy works
- Goal: not to show you everything Pandas can do, rather just what we need for ML / data science
- If you have a question about something not covered, just ask!
- Most times: Load in data and immediately convert it into Numpy array
- Most features you won't use often, you'll just forget them

```
In [5]: X = np.array(X)

In [6]: X.shape
Out[6]: (100, 3)

In [7]: import pandas as pd

In [8]: X = pd.read_csv("data_2d.csv", header=None)

In [9]: type(X)
Out[9]: pandas.core.frame.DataFrame

In [10]: X.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100 entries, 0 to 99
Data columns (total 3 columns):
0    100 non-null float64
1    100 non-null float64
2    100 non-null float64
dtypes: float64(3)
memory usage: 3.1 KB
```

```
0  17.930201  94.520592  320.259530
1  97.144697  69.593282  404.634472
2  81.775901   5.737648  181.485108
3  55.854342  70.325902  321.773638
4  49.366550  75.114040  322.465486

In [12]: X.head(10)
Out[12]:
           0          1           2
0  17.930201  94.520592  320.259530
1  97.144697  69.593282  404.634472
2  81.775901   5.737648  181.485108
3  55.854342  70.325902  321.773638
4  49.366550  75.114040  322.465486
5   3.192702  29.256299   94.618811
6  49.200784  86.144439  356.348093
7  21.882804  46.841505  181.653769
8  79.509863  87.397356  423.557743
9  88.153887  65.205642  369.229245
```

```
In [11]: X.head()
Out[11]:
           0          1           2
0  17.930201  94.520592  320.259530
1  97.144697  69.593282  404.634472
2  81.775901   5.737648  181.485108
3  55.854342  70.325902  321.773638
4  49.366550  75.114040  322.465486

In [12]: X.head(10)
Out[12]:
           0          1           2
0  17.930201  94.520592  320.259530
1  97.144697  69.593282  404.634472
2  81.775901   5.737648  181.485108
3  55.854342  70.325902  321.773638
4  49.366550  75.114040  322.465486
5   3.192702  29.256299   94.618811
6  49.200784  86.144439  356.348093
7  21.882804  46.841505  181.653769
8  79.509863  87.397356  423.557743
9  88.153887  65.205642  369.229245
```

```
In [14]: M = X.as_matrix()

In [15]: type(M)
Out[15]: numpy.ndarray
```

```
In [17]: X.head()
Out[17]:
           0          1           2
0  17.930201  94.520592  320.259530
1  97.144697  69.593282  404.634472
2  81.775901   5.737648  181.485108
3  55.854342  70.325902  321.773638
4  49.366550  75.114040  322.465486
```

Numpy: X[0] -> 0th row
Pandas: X[0] -> column that has name 0

```
In [18]: type(X[0])
Out[18]: pandas.core.series.Series

In [19]: X.iloc[0]
Out[19]:
0     17.930201
1     94.520592
2    320.259530
Name: 0, dtype: float64

In [20]: X.ix[0]
Out[20]:
0     17.930201
1     94.520592
2    320.259530
Name: 0, dtype: float64

In [21]: type(X.ix[0])
Out[21]: pandas.core.series.Series
```

```
In [23]: X[ X[0] < 5 ]
Out[23]:
           0          1           2
5   3.192702  29.256299   94.618811
44  3.593966  96.252217  293.237183
54  4.593463  46.335932  145.818745
90  1.382983  84.944087  252.905653
99  4.142669  52.254726  168.034401
```

```
n [1]: import pandas as pd

n [2]: df = pd.read_csv("international-airline-passengers.csv", engine="python", skipfooter=3)

n [3]: df.columns
ut[3]: Index([u'Month', u'International airline passengers: monthly totals in thousands. Jan 49 ?
ec 60'], dtype='object')

n [4]: df.columns = ["month", "passengers"]

n [5]: df.columns
ut[5]: Index([u'month', u'passengers'], dtype='object')
```

```
In [8]: df['ones'] = 1

In [9]: df.head()
Out[9]:
     month  passengers  ones
0  1949-01         112     1
1  1949-02         118     1
2  1949-03         132     1
3  1949-04         129     1
4  1949-05         121     1
```

The Apply Function

- What if we want to assign a new column value where each cell is derived from the values already in its row?
- Ex. Model interaction between X1 and X2 → X1*X2
- We use the apply function!

```
df['x1x2'] = df.apply(lambda row: row['x1']*row['x2'], axis=1)
```

- Pass in axis=1 so the function gets applied across each row instead of each column
- Think of it like Python's map function

The Apply Function

- If you're not familiar with lambda, this is equivalent:

```
def get_interaction(row):
 return row['x1'] * row['x2']
```

```
df['x1x2'] = df.apply(get_interaction, axis=1)
```

- Function you pass in takes 1 argument, the row

The Apply Function

- Equivalent to:

```
interactions = []
for idx, row in df.iterrows():
  x1x2 = row[‘x1’] * row[‘x2’]
  interactions.append(x1x2)
df[‘x1x2’] = interactions
```

- Never actually do this

```
In [10]: from datetime import datetime

In [11]: datetime.strptime("1949-05", "%Y-%m")
Out[11]: datetime.datetime(1949, 5, 1, 0, 0)

In [12]: df['dt'] = df.apply(lambda row: datetime.strptime(row['month'], "%Y-%m"), axis=1)

In [13]: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 144 entries, 0 to 143
Data columns (total 4 columns):
month         144 non-null object
passengers    144 non-null int64
ones          144 non-null int64
dt            144 non-null datetime64[ns]
dtypes: datetime64[ns](1), int64(2), object(1)
memory usage: 5.6+ KB
```

```
In [1]: import pandas as pd

In [2]: t1 = pd.read_csv('table1.csv')

In [3]: t2 = pd.read_csv('table2.csv')

In [4]:
```

```
In [6]: m = pd.merge(t1, t2, on='user_id')

In [7]: m
Out[7]:
    user_id              email  age  ad_id  click
0         1    alice@gmail.com   20      1      1
1         1    alice@gmail.com   20      2      0
2         1    alice@gmail.com   20      5      0
3         2      bob@gmail.com   25      3      0
4         2      bob@gmail.com   25      4      1
5         2      bob@gmail.com   25      1      0
6         3    carol@gmail.com   30      2      0
7         3    carol@gmail.com   30      1      0
8         3    carol@gmail.com   30      3      0
9         3    carol@gmail.com   30      4      0
10        3    carol@gmail.com   30      5      1
```

```
In [8]: t1.merge(t2, on='user_id')
Out[8]:
    user_id            email  age  ad_id  click
0         1  alice@gmail.com   20      1      1
1         1  alice@gmail.com   20      2      0
2         1  alice@gmail.com   20      5      0
3         2    bob@gmail.com   25      3      0
4         2    bob@gmail.com   25      4      1
5         2    bob@gmail.com   25      1      0
6         3  carol@gmail.com   30      2      0
7         3  carol@gmail.com   30      1      0
8         3  carol@gmail.com   30      3      0
9         3  carol@gmail.com   30      4      0
10        3  carol@gmail.com   30      5      1
```

Loading in Data

- Deep learning / machine learning learns from data - so you need data loading to be an automatic reflex
- Unstructured data - the Internet
- Semi-structured data - Apache logs

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326
```

- Structured data - Kaggle and other datasets (usually CSV)
- CSV = comma separated values
- Each row is a record
- Each record's values are separated by commas
- It's a table, so you can open in Excel
- But data scientists like matrices, so we'll turn it into a matrix of numbers (manually first)
- File: data_2d.csv Folder: linear_regression_class

MATPLOTLIB

```
In [3]: import matplotlib.pyplot as plt

In [4]: x = np.linspace(0, 10, 10)

In [5]: y = np.sin(x)

In [6]: plt.plot(x, y)
Out[6]: [<matplotlib.lines.Line2D at 0x10bc518d0>]

In [7]: plt.show()
```

```
In [3]: import matplotlib.pyplot as plt

In [4]: x = np.linspace(0, 10, 10)

In [5]: y = np.sin(x)

In [6]: plt.plot(x, y)
Out[6]: [<matplotlib.lines.Line2D at 0x10bc518d0>]

In [7]: plt.show()

In [8]: plt.plot(x, y)
Out[8]: [<matplotlib.lines.Line2D at 0x10d416c90>]

In [9]: plt.xlabel("Time")
Out[9]: <matplotlib.text.Text at 0x10bc85090>

In [10]: plt.ylabel("Some function of time")
Out[10]: <matplotlib.text.Text at 0x10d9e8c50>

In [11]: plt.title("My cool chart")
Out[11]: <matplotlib.text.Text at 0x10d3e3b90>
```

```
Out[11]: <matplotlib.text.Text at 0x10d3e3b90>

In [12]: plt.show()

In [13]: x = np.linspoce(0, 10, 100)

In [14]: y = np.sin(x)

In [15]: plt.s
```

SCIPY

Scipy

- Gaussian PDF:

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\sigma^2\pi}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- Don't we already have all the tools we need to calculate this? Square, divide, exponential, square root
- Scipy is faster

```
In [20]: from scipy.stats import norm

In [21]: norm.pdf(0)
Out[21]: 0.3989422804014327

In [22]: norm.pdf(0, loc=5, scale=10)
Out[22]: 0.035206532676429952

In [23]: import numpy as np

In [24]: r = np.random.randn(10)

In [25]: norm.pdf(r)
Out[25]:
array([ 0.3773593 ,  0.14510843,  0.17362849,  0.39845
        0.06699539,  0.13945862,  0.27778476,  0.39370

In [26]:
```

Log Pdf

Joint probability vs. log of joint probability of data samples (+ faster than *):

$$p(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} p(x_i)$$

$$logp(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} logp(x_i)$$

Log of Gaussian PDF (much faster since no exponential!):

$$logp(x) = \; -\frac{1}{2}log(2\pi\sigma^2) - \frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}$$

```
In [26]: norm.logpdf(r)
Out[26]:
array([-0.97455749, -1.93027406, -1.7508374 , -0.92014871, -0.97586019,
       -2.70313148, -1.96998735, -1.28090872, -0.93215281, -0.94569553])

In [27]:
```

```
In [27]: norm.cdf(r)
Out[27]:
array([ 0.36936958,  0.92251683,  0.09854567,  0.51961891,  0.63209485,
        0.97055524,  0.92645139,  0.80257278,  0.43542906,  0.40852887])
```

```
In [27]: norm.cdf(r)
Out[27]:
array([ 0.36936958,  0.92251683,  0.09854567,  0.51961891,  0.63209485,
        0.97055524,  0.92645139,  0.80257278,  0.43542906,  0.40852887])
```

```
In [28]: norm.logcdf(r)
Out[28]:
array([-0.99595755, -0.08064966, -2.31723518, -0.65465961, -0.45871582,
       -0.02988696, -0.0763937 , -0.21993273, -0.8314234 , -0.8951927 ])
```

```
In [45]: cov = np.array([[1, 0.8], [0.8, 3]])

In [46]: from scipy.stats import multivariate_normal as mvn

In [47]: mu = np.array([0,2])

In [48]: r = mvn.rvs(mean=mu, cov=cov, size=1000)

In [49]: plt.scatter(r[:,0], r[:,1])
Out[49]: <matplotlib.collections.PathCollection at 0x10bfb3790>

In [50]: plt.axis('equal')
Out[50]: (-5.0, 5.0, -6.0, 8.0)

In [51]: plt.show()

In [52]: r = np.random.multivariate_normal(mean=mu, cov=cov, size=1000)

In [53]:
```