クレジット：
UTokyo Online Education Education コンピュータシステム概論 2018 小林克志

ライセンス：
利用者は、本講義資料を、教育的な目的に限ってページ単位で利用することができます。特に記載のない限り、本講義資料はページ単位でクリエイティブ・コモンズ 表示-非営利-改変禁止 ライセンスの下に提供されています。
http://creativecommons.org/licenses/by-nc-nd/4.0/

本講義資料内には、東京大学が第三者より許諾を得て利用している画像等や、各種ライセンスによって提供されている画像等が含まれています。個々の画像等を本講義資料から切り離して利用することはできません。個々の画像等の利用については、それぞれの権利者の定めるところに従ってください。

NumPy

この資料は The Python Tutorial (https://docs.python.org/3.6/tutorial/index.html#the-python-tutorial) (日本語版 (https://docs.python.jp/3/tutorial/)) および Python for Data Analysis:Wrangling with Pandas, Numpy and IPython (http://shop.oreilly.com/product/0636920050896.do)を参考に作成した。

NumPy(Numerical Python) は Python の数値計算にとって重要なパッケージの一つ。

NumPy は多次元配列、配列全体にわたる数値計算、ディスクの読み書きやメモリマップドファイル、線形代数、乱数生成、フーリエ変換、C, C++, FORTRAN ライブラリへの API などを含む。

Numpy はモジュールを読み込んで利用する:

```
import numpy as np
```

多次元配列

NumPy で最も重要なクラス、`np.ndarray`、は多次元配列を扱うクラスである。

`np.ndarray` の生成方法はいくつかある:

- `np.ndarray()` は要素が初期化されないことに注意すること。
  `np.array()`, `np.zeros()` の方が有用
- `np.zeros()` は全てが 0 の、`np.ones()` は全てが 1 の配列を生成する。
- `np.array()` では
- この後の説明では、乱数を生成する `np.random.randn()` を多く利用する。
  `np.random` の説明は後述する

 CC BY-NC-ND

In [185]:

```
import numpy as np

print("numpy.ndarray():")
data0 = np.ndarray([4,4])          # 次元要素の数をリストで与える、ここ
では 4 x 4 の行列
print(data0)
print()

print("Sample Vector:")
data1 = np.zeros(4)                # 長さ 4 のベクトル（1次元配列），
ゼロで初期化
print(data1)                       # 確認
print("Dimension:", data1.ndim)    # 次元を知りたい
print("Shape:", data1.shape)       # サイズ
print("Type:", data1.dtype)        # 要素の型
print()

print("Sample Matrix:")
data2 = np.zeros([2,3])            # 2 x 3 の行列を作る, ゼロで初期化
print(data2)                       # 行列の確認
print("Dimension:", data2.ndim)    # 次元を知りたい
print("Shape:", data2.shape)       # サイズ
print("Type:", data2.dtype)        # 要素の型
print()

print("Sample Tensor:")
data3 = np.ones([3 ,3, 3])         # 3 x 3 x 3 のテンソルを作る
print(data3)                       # 行列の確認
print("Dimension:", data3.ndim)    # 次元を知りたい
print("Shape:", data3.shape)       # サイズ
print("Type:", data3.dtype)        # 要素の型
```

UTokyo Online Education Education コンピュータシステム概論 2018 小林克志 CC BY-NC-ND

```
numpy.ndarray():
[[2.22044605e-16 4.44089210e-16 5.55111512e-17 2.22044605e-16]
 [1.66533454e-16 4.44089210e-16 3.33066907e-16 2.22044605e-16]
 [1.24900090e-16 1.11022302e-16 2.22044605e-16 1.11022302e-16]
 [2.22044605e-16 1.11022302e-16 4.44089210e-16 1.66533454e-16]]

Sample Vector:
[0. 0. 0. 0.]
Dimension: 1
Shape: (4,)
Type: float64

Sample Matrix:
[[0. 0. 0.]
 [0. 0. 0.]]
Dimension: 2
Shape: (2, 3)
Type: float64

Sample Tensor:
[[[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.]]]
Dimension: 3
Shape: (3, 3, 3)
Type: float64
```

すでにあるリストあるいはタプルから `np.ndarray` を生成するには、`np.array()` を使う:

 CC BY-NC-ND

```
In [190]: import numpy as np
print("Sample Vector")
data1 = np.array([1, 2, 3])        # リストから作る
print(type(data1))
print(data1)
print("Dimension:", data1.ndim)    # 次元を知りたい
print("Shape:", data1.shape)       # サイズ
print("Type:", data1.dtype)        # 要素の型
print()
print("Sample Matrix")
data2 = np.array([[1, 2, 3],[4, 5, 6], [7, 8, 9]])    # 2 次元リストでも同様
print(data2)
print("Dimension:", data2.ndim)    # 次元を知りたい
print("Shape:", data2.shape)       # サイズ
print("Type:", data2.dtype)        # 要素の型
print()
```

```
Sample Vector
<class 'numpy.ndarray'>
[1 2 3]
Dimension: 1
Shape: (3,)
Type: int64

Sample Matrix
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Dimension: 2
Shape: (3, 3)
Type: int64
```

`np.random.randn()` も `np.ndarray` を生成する。
ただし引数は次元の要素数を可変長引数として与える（リストではない）ことに注意:

```
In [37]: import numpy as np
np.random.randn(3,3,3)      # 3 x 3 x 3 のテンソル
```

```
Out[37]: array([[[ 0.91738216, -1.5244501 , -1.94756131],
        [-1.26215377, -0.32747145, -1.08663433],
        [ 0.79725466,  0.04450173,  0.34936664]],

       [[ 1.0146844 , -0.15195247, -1.32524809],
        [ 0.76273512, -0.50376583,  1.78496722],
        [ 0.62122625, -0.29175216,  0.04721141]],

       [[-1.57382761, -0.07732282,  0.03008112],
        [-0.67719275, -0.13127505, -1.17161919],
        [ 1.46676563, -0.22218116,  0.03758074]]])
```

`numpy.arange()` は Numpy 版の `range()` 関数で、少数点以下のステップにも対応している。 リストではなく `np.ndarray` が生成されることに注意:

In [54]:
```
import numpy as np
print(type(np.arange(10)))      # ndarray を返している
print(np.arange(10))            # 組み込み range() 関数と変わらない
print(np.arange(1, step=0.1))   # 0.1 刻み
```

```
<class 'numpy.ndarray'>
[0 1 2 3 4 5 6 7 8 9]
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

ndarray の算法

`np.ndarray` の演算子の性質は以下のとおり。多くの演算が要素単位に行われることに注意する:

In [43]:
```
import numpy as np
data = np.array([[1, 2, 3],[ 4, 5, 6]])
print("Base matrix:")
print(data)
print("10 * data:")
print(10*data)                      # スカラー倍
print("10 * data + data:")
print(10*data + data)               # 要素の加算

print("10 * data - data:")
print(10*data - data)               # 要素の差
print("data * data:")
print(data * data)                  # 要素ごとの積（配列演算ではないことに
注意）

print("1 / data:")
print(1 / data)                     # 商
print("data ** 0.5:")
print(data**0.5)                    # べき乗

print("data == 2:")
print(data == 2)                    # スカラーとの比較
print("data != 2:")
print(data != 2)                    # スカラーとの比較2
print("~(data == 2):")
print(~(data == 2))                 # スカラーとの比較3
print("data > 2:")
print(data > 2)                     # スカラーとの比較4
```

UTokyo Online Education Education コンピュータシステム概論 2018 小林克志 CC BY-NC-ND

```
Base matrix:
[[1 2 3]
 [4 5 6]]
10 * data:
[[10 20 30]
 [40 50 60]]
10 * data + data:
[[11 22 33]
 [44 55 66]]
10 * data - data:
[[ 9 18 27]
 [36 45 54]]
data * data:
[[ 1  4  9]
 [16 25 36]]
1 / data:
[[1.         0.5        0.33333333]
 [0.25       0.2        0.16666667]]
data ** 0.5:
[[1.         1.41421356 1.73205081]
 [2.         2.23606798 2.44948974]]
data == 2:
[[False  True False]
 [False False False]]
data != 2:
[[ True False  True]
 [ True  True  True]]
~(data == 2):
[[ True False  True]
 [ True  True  True]]
data > 2:
[[False False  True]
 [ True  True  True]]
```

np.ndarray 同士の比較演算は:

In [44]:
```
import numpy as np
data1 = np.array([[1, 2, 3],[ 4, 5, 6]])
data2 = np.array([[3, 2, 1],[ 6, 5, 4]])

print("data1 < data2:")
print(data1 < data2)          # np.ndarray 同士の比較演算
```

```
[[ True False False]
 [ True False False]]
```

単なる代入では np.ndarray の要素そのものはコピーされないことに注意（NumPy に限ったことではないが）：

 CC BY-NC-ND

```
In [47]: import numpy as np
         print("Substitution:")
         data = np.array([1, 2, 3, 4, 5, 6])
         data1 = data[2:4]                        # 代入なのでデータはコピーされな
         い
         data1[1] = 1234
         print(data)                              # data も書き換えられる

         print("Explicit copy:")
         data = np.array([1, 2, 3, 4, 5, 6])
         data1 = data[2:4].copy()                 # コピーが必要な場合はこちらを使
         う
         data1[1] = 1234
         print(data)
```

```
Substitution:
[   1    2    3 1234    5    6]
Explicit copy:
[1 2 3 4 5 6]
```

多次元 np.ndarray の要素はインデックス（添字）を使ってアクセスできる。
インデックスの与え方は 2 次元リスト、コンマ区切りのいずれも可能:

```
In [57]: import numpy as np
         data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

         print(data[0])                  # 1 行目
         print(data[1][2])               # 2 行、3列目
         print(data[1,2])                # こちらでも同じ
```

```
[1 2 3]
6
6
```

スライス

リスト同様にスライス表記も使える。スライスへのスカラー値の代入は該当するすべての要素が更新される:

```
In [64]: import numpy as np
         data = np.array([1, 2, 3, 4, 5, 6])

         print(data[2:4])

         data[2:4] = 7             # スライスに代入してみる
         print(data)
```

```
[3 4]
[1 2 7 7 5 6]
```

多次元配列でのスライス表記は:

 CC BY-NC-ND

In [62]:
```
import numpy as np
data = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13
, 14, 15, 16]])

print(data[:2])                # 1-2 行
print(data[:,:2])              # 1-2 列

print(data[0:2,2:4])           # 1-2 行,3-4 列
print(data[:2,2:])             # 上に同じ
```

```
[[1 2 3 4]
 [5 6 7 8]]
[[ 1  2]
 [ 5  6]
 [ 9 10]
 [13 14]]
[[3 4]
 [7 8]]
[[3 4]
 [7 8]]
```

Bool インデックス

`np.ndarray` における Bool インデックスの使用例を示した。 ここで、`data` の各行は、`names` の要素に対応している:

 CC BY-NC-ND

```
In [65]: import numpy as np
names = np.array(["apple", "orange", "grape", "apple", "apple", "
kiwi", "orange"])
data = np.zeros((7,3))
for i in range(7):
    for j in range(3):
        data[i,j] = i + j
print(data)
print()

print(names=="apple")              # apple と一致する bool リストを取
得する
print()

print(data[names=="apple"])        # apple に対応する行だけ取り出す
print()

print(data[names=="apple", 1:])    # スライス表記と組み合わせも可能

print(data[(names=="apple") | (names =="kiwi")]) # apple と kiwi
行をとりだす
```

```
[[0. 1. 2.]
 [1. 2. 3.]
 [2. 3. 4.]
 [3. 4. 5.]
 [4. 5. 6.]
 [5. 6. 7.]
 [6. 7. 8.]]

[ True False False  True  True False False]

[[0. 1. 2.]
 [3. 4. 5.]
 [4. 5. 6.]]

[[1. 2.]
 [4. 5.]
 [5. 6.]]
[[0. 1. 2.]
 [3. 4. 5.]
 [4. 5. 6.]
 [5. 6. 7.]]
```

Bool インデックスは多次元配列でも利用できる:

 CC BY-NC-ND

```
In [76]: import numpy as np
         data = np.random.randn(3,7)

         print(data)
         print()
         print(data < 0)
         print()
         data[data < 0] = 0          # 負の要素のみ置き換え
         print(data)
```

```
[[-0.12120045 -0.59257139  1.21046481 -0.99058345 -1.62998535 -1.
30219124
  -1.64351472]
 [ 2.26759974  1.83176788  0.83085159 -0.79055313 -0.48373395 -0.
05915556
   0.83655388]
 [ 0.50419437 -1.04214763  1.69827135  0.02270963  0.69686814  0.
53176455
  -0.38752502]]

[[ True  True False  True  True  True  True]
 [False False False  True  True  True False]
 [False  True False False False False  True]]

[[ 0.          0.          1.21046481  0.          0.          0.
0.        ]
 [ 2.26759974  1.83176788  0.83085159  0.          0.          0.
   0.83655388]
 [ 0.50419437  0.          1.69827135  0.02270963  0.69686814  0.
53176455
   0.        ]]
```

Fancy インデックス

`np.ndarray` では配列の添字をリストで与え、一部の取り出しおよび並べ替えもできる:

 CC BY-NC-ND

In [196]:
```
import numpy as np
data = np.zeros((7,3))
for i in range(7):
    for j in range(3):
        data[i,j] = i + j/10
print("Sample:")
print(data)
print()
print("Pick 7, 5, 1 rows:")
print(data[[6,0,4]])        # 7, 1, 5 行目を取り出し、並べる、インデ
ックスが 2 重リストになっていることに注意
print()

print("Negative index:")
print(data[[-3,2]])        #負の添字も使える
```

```
Sample:
[[0.  0.1 0.2]
 [1.  1.1 1.2]
 [2.  2.1 2.2]
 [3.  3.1 3.2]
 [4.  4.1 4.2]
 [5.  5.1 5.2]
 [6.  6.1 6.2]]

Pick 7, 5, 1 rows:
[[6.  6.1 6.2]
 [0.  0.1 0.2]
 [4.  4.1 4.2]]

Negative index:
[[4.  4.1 4.2]
 [2.  2.1 2.2]]
```

複数の添字リストによって取り出す（行、列）の指定も可能である。結果は一次元リストになる:

 CC BY-NC-ND

```
In [199]: import numpy as np
data = np.zeros((7,3))
for i in range(7):
    for j in range(3):
        data[i,j] = i + j/10

print("Sample:")
print(data)
print()
print("Pick data[0,2], data[2,0], data[4,1]:")
print(data[[0,2,4], [2,0,1]])  # [data[0,2], data[2,0], data[4,1
]] と等価
```

```
Sample:
[[0.  0.1 0.2]
 [1.  1.1 1.2]
 [2.  2.1 2.2]
 [3.  3.1 3.2]
 [4.  4.1 4.2]
 [5.  5.1 5.2]
 [6.  6.1 6.2]]

Pick data[0,2], data[2,0], data[4,1]:
[0.2 2.  4.1]
```

以下の例のように Funcy インデックスを連続して適用することも可能である:

```
In [94]: import numpy as np
data = np.zeros((7,3))
for i in range(7):
    for j in range(3):
       data[i,j] = i + j/10

print(data)
print()

print(data[[1,3,5]][:,[2,1,0]])      # 2, 4, 6 行を取り出したあとで
、列を 3, 2, 1 の順に並び替える。
```

```
[[ 0.   0.1  0.2]
 [ 1.   1.1  1.2]
 [ 2.   2.1  2.2]
 [ 3.   3.1  3.2]
 [ 4.   4.1  4.2]
 [ 5.   5.1  5.2]
 [ 6.   6.1  6.2]]

[[ 1.2  1.1  1. ]
 [ 3.2  3.1  3. ]
 [ 5.2  5.1  5. ]]
```

転置行列

ndarray の転置は、`ndaray.T` で得られる:

```
In [203]: import numpy as np
          data = np.zeros((7,3))
          for i in range(7):
              for j in range(3):
                  data[i,j] = i + j/10

          print("Sample:")
          print(data)
          print()

          print("Transpose:")
          print(data.T)                    # 行列の転置（関数でないことに注意）
```

```
Original:
[[0.  0.1 0.2]
 [1.  1.1 1.2]
 [2.  2.1 2.2]
 [3.  3.1 3.2]
 [4.  4.1 4.2]
 [5.  5.1 5.2]
 [6.  6.1 6.2]]

Transpose:
[[0.  1.  2.  3.  4.  5.  6. ]
 [0.1 1.1 2.1 3.1 4.1 5.1 6.1]
 [0.2 1.2 2.2 3.2 4.2 5.2 6.2]]
```

汎用関数

`np.exp()`, `np.sqrt()` などは `np.ndarray` 要素に対して実行される。
`np.maximum()`は引数に与えられる二つの `np.ndarray` の同じ添字をもつ要素の最大値で構成される `np.ndarray` を
`np.modf()` は各要素の小数部、整数部の `np.ndarray` をリストで返す:

 CC BY-NC-ND

In [209]:
```
import numpy as np
data1 = np.random.randn(4,4)*4
data2 = np.random.randn(4,4)*4
print("Sample:")
print(data1, data2)
print()

print("np.maxumim():")
print(np.maximum(data1, data2))    # 大きい方の要素を取り出す
print()

print("np.modf():")
print(np.modf(data1))              # 整数・小数部に分けたい
print()

print("np.sqrt() with reals:")
print(np.sqrt([-2, -1, 0, 1, 2, 3]))       # 虚数解は Not a Numbe
r (NaN)
print()
print("np.sqrt() with complexes:")
print(np.sqrt([-2 + 0J, -1, 0, 1, 2, 3]))  # 虚数解も欲しいときは、n
darray を複素数型に
```

 CC BY-NC-ND

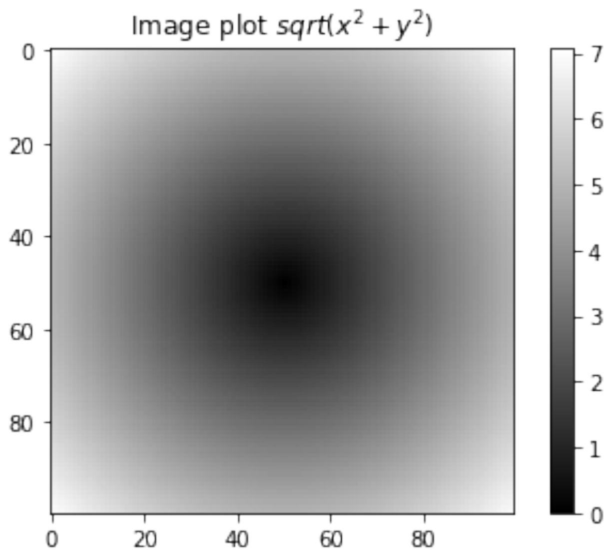
```
Sample:
[[-2.85574802 -5.85855744  3.1294711   5.33592706]
 [-0.58282775  1.97001317 -3.25784614 -5.30399032]
 [-1.01484877  0.2026208   0.12232135 -0.02411796]
 [-1.37562422  4.01701034 -1.70257343  2.74811638]] [[ 2.23770787
-2.11596678 -0.36781663  3.39674859]
 [ 6.32293161 -1.58634096 -1.85988616  1.55575025]
 [ 0.98451453  1.31198565 -1.67222442  2.28948338]
 [ 4.14208928  4.32074424 -6.8168172   6.12551588]]

np.maxumim():
[[ 2.23770787 -2.11596678  3.1294711   5.33592706]
 [ 6.32293161  1.97001317 -1.85988616  1.55575025]
 [ 0.98451453  1.31198565  0.12232135  2.28948338]
 [ 4.14208928  4.32074424 -1.70257343  6.12551588]]

np.modf():
(array([[-0.85574802, -0.85855744,  0.1294711 ,  0.33592706],
       [-0.58282775,  0.97001317, -0.25784614, -0.30399032],
       [-0.01484877,  0.2026208 ,  0.12232135, -0.02411796],
       [-0.37562422,  0.01701034, -0.70257343,  0.74811638]]), ar
ray([[-2., -5.,  3.,  5.],
       [-0.,  1., -3., -5.],
       [-1.,  0.,  0., -0.],
       [-1.,  4., -1.,  2.]]))

np.sqrt() with reals:
[        nan        nan 0.         1.         1.41421356 1.7320508
1]

np.sqrt() with complexes:
[0.        +1.41421356j 0.        +1.        j 0.        +0.
j
 1.        +0.        j 1.41421356+0.        j 1.73205081+0.
j]
/Users/ikob/anaconda3/lib/python3.6/site-packages/ipykernel_launc
her.py:17: RuntimeWarning: invalid value encountered in sqrt
```

配列志向のプログラミング

Numpy を利用すれば（他ではループが必要な）配列表記のまま計算を実行することができる。
平面直交座標系の点 (x, y) の中心 (0,0)からの距離を濃淡で表現してみる:

 CC BY-NC-ND

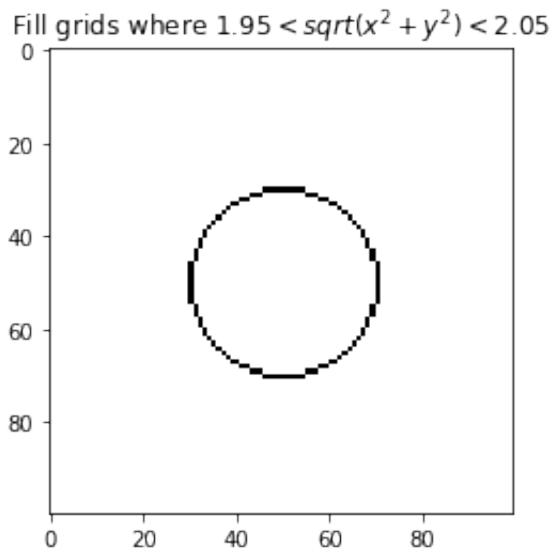
In [98]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

xs, ys = np.meshgrid(np.arange(-5,5, 0.1), np.arange(-5,5, 0.1))
z = np.sqrt(xs **2 + ys **2)

plt.imshow(z, cmap=plt.cm.gray)
plt.colorbar()
plt.title("Image plot $sqrt(x^2 + y^2)$")
plt.show()
```



条件付きロジック

`np.where(条件、真のときの値、偽のときの値)` は三項表記 `x if 条件 else y` の `np.ndarray` 版である。
直前に説明した、原点からの距離に応じた濃淡の問題を利用して説明する。 以下の例では、100 x 100 に分割した平面で原点からの距離が (1.95, 2.05) の範囲であれば 0 を、それ以外は 10 を与えている:

In [99]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

xs, ys = np.meshgrid(np.arange(-5,5, 0.1), np.arange(-5,5, 0.1))
z = np.sqrt(xs **2 + ys **2)

plt.imshow(np.where((1.95 < z) & (z < 2.05), 0, 10), cmap=plt.cm.gray)
plt.title("Fill grids where $1.95 < sqrt(x^2 + y^2) < 2.05 $")
plt.show()
```



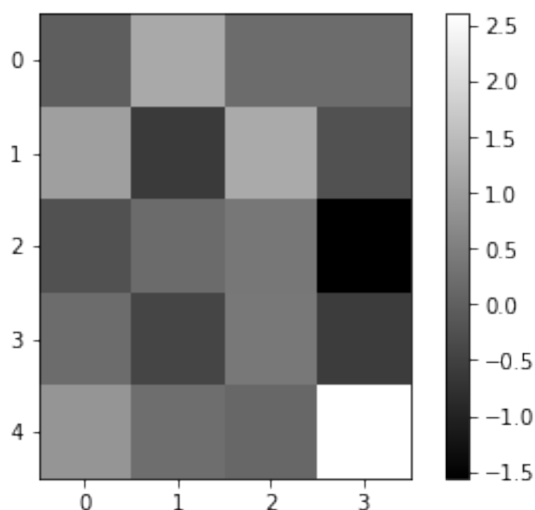
統計

`np.ndarray` の統計量、総和、平均、標準偏差(`np.sum()`, `np.mean()`, `np.std()`)、を計算する関数も用意されている。

CC BY-NC-ND

```
In [110]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

data = np.random.randn(5,4)

print("Data:")
print(data)

print("Mean(All):", data.mean())
print("Total(All):", data.sum())

print("Mean(column wise):", data.mean(axis=0))          # タテ方向の平均をリストで返す
print("Mean(row wise):",data.mean(axis=1))              # 横方向の平均

plt.imshow(data, cmap=plt.cm.gray)                      # 意味はないが可視化してみる
plt.colorbar()
plt.show()
```

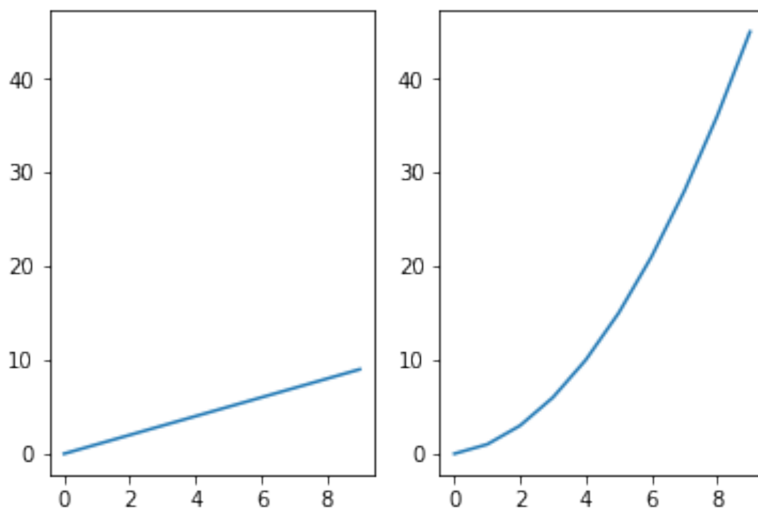
```
Data:
[[-0.02078304  1.19900967  0.20116583  0.21196276]
 [ 1.03140621 -0.6125166   1.22208293 -0.22873657]
 [-0.23975873  0.18968221  0.38595628 -1.56069793]
 [ 0.20461808 -0.43282198  0.40910466 -0.57456353]
 [ 0.87426994  0.24254426  0.11736884  2.60864398]]
Mean(All): 0.2613968628500536
Total(All): 5.227937257001072
Mean(column wise): [0.36995049 0.11717951 0.46713571 0.09132174]
Mean(row wise): [ 0.3978388   0.35305899 -0.30620454 -0.09841569
0.96070675]
```



累積和の計算には `np.cumsum()` が利用できる:

In [119]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

data = np.arange(10)
print("Cumulative sum:")
print(data.cumsum())

ax = plt.subplot(1,2,1)
plt.plot(data)

plt.subplot(1,2,2, sharey=ax)
plt.plot(data.cumsum())

plt.show()
```

```
Cumulative sum:
[ 0  1  3  6 10 15 21 28 36 45]
```



ブール演算

ブール演算では `True` / `False` を返す。 これを利用して、ndarray 中の `True` を数えるのは:

In [123]:
```
import numpy as np
data = np.random.randn(100)

print((data > 0).sum())        # 0 より大きい要素の数を数える
```

```
49
```

さらに、`numpy.any()` および `numpy.all` メソッドは ndarray にわたって論理値を検査する。 これらのメソッドでは論理値以外でも動作し、この場合 0 以外の値は `True` と評価される。

 CC BY-NC-ND

In [130]:
```
data = np.array([False, True, True, True])
print("data:",data)
print("data.all():", data.all())
print("data.any():", data.any())
print("")
data = np.array([0,1,2,3,4,5])
print("data:",data)
print("data.all():", data.all())
print("data.any():", data.any())
```

```
data: [False  True  True  True]
data.all(): False
data.any(): True

data: [0 1 2 3 4 5]
data.all(): False
data.any(): True
```

ソート

list と同様に `np.sort()` によってソート（並べ替え）をおこなう。
`np.sort()` の第一引数にソート対象の次元を与えることもできる。
キーワード引数 `kind` でソートアルゴリズムの指定もできる。
逆順にしたい場合は、`new = old[::-1]`とすれば良い。

UTokyo Online Education Education コンピュータシステム概論 2018 小林克志 CC BY-NC-ND

In [136]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

data = np.random.randn(5)
print("Sample:")
print(data)

print("Sorted:")
data.sort()
print(data)

print("Reverse:")
data = data[::-1]                # ソート結果を逆順にする
print(data)

data = np.random.randn(5,4)      # 2 次元
print("2D Sample:")
print(data)

print("Sorted without axis:")
data.sort()                      # 次元（軸）が与えられない場合はもっ
とも低い次元でソート
print(data)                      # この場合、行方向でソート
print()

print("Sorted with axis:")
data.sort(0)
print(data)
```

```
Sample:
[ 1.63152519  0.90061293  1.51034952  0.24006905 -0.99878574]
Sorted:
[-0.99878574  0.24006905  0.90061293  1.51034952  1.63152519]
Reverse:
[ 1.63152519  1.51034952  0.90061293  0.24006905 -0.99878574]
2D Sample:
[[-0.0045587  -0.17275276 -0.05330216 -0.4212952 ]
 [-2.03376098  0.07177864 -1.04034823  0.91599479]
 [-0.56874863 -0.84804534 -1.00323581  0.97183046]
 [ 0.73028927 -0.17050413  0.24199593 -0.10922277]
 [ 0.26836155 -0.14156026  1.08342267 -1.28399945]]
Sorted without axis:
[[-0.4212952  -0.17275276 -0.05330216 -0.0045587 ]
 [-2.03376098 -1.04034823  0.07177864  0.91599479]
 [-1.00323581 -0.84804534 -0.56874863  0.97183046]
 [-0.17050413 -0.10922277  0.24199593  0.73028927]
 [-1.28399945 -0.14156026  0.26836155  1.08342267]]

Sorted with axis:
[[-2.03376098 -1.04034823 -0.56874863 -0.0045587 ]
 [-1.28399945 -0.84804534 -0.05330216  0.73028927]
 [-1.00323581 -0.17275276  0.07177864  0.91599479]
 [-0.4212952  -0.14156026  0.24199593  0.97183046]
 [-0.17050413 -0.10922277  0.26836155  1.08342267]]
```

 CC BY-NC-ND

Unique など

1次元 ndarray 向けの関数として、`np.unique(ndarray)` もよく使われる。 これは、ソート済みのユニークな値を ndarray として返す。 また、`np.in1d(ndarray, 要素のリスト)` は ndarray の要素ごとに第2引数のリストの有無を検査し、結果を ndarray として返す:

In [140]:

```
import numpy as np

data = np.array(["apple", "banana", "orange", "kiwi", "apple", "
banana"])

print("np.unique():")
print(np.unique(data))
print(np.array(sorted(set(data))))    # 集合(セット)を使えば等価にな
る

print("np.in1d():")
print(np.in1d(data, ["apple", "kiwi"]))
```

```
np.unique():
['apple' 'banana' 'kiwi' 'orange']
['apple' 'banana' 'kiwi' 'orange']
np.in1d():
[ True False False  True  True False]
```

線形代数

NumPy では行列積（Innter Product）は `np.dot()` をつかう。 NumPy で二項演算子 `*` は要素同士の積になることに注意、どうしても 演算子を使いたい場合は `@` が用意されている:

 CC BY-NC-ND

In [148]:
```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]])
a2 = np.array([[3, 4, 5], [6, 7, 8]])
b = np.array ([[1,2],[3,4],[5,6]])

print("Inner product:")
print(a.dot(b))
print("Another Inner product:")
print(np.dot(a, b))       # 上と等価
print("Inner product with @ operator:")
print(a @ b)              # 行列積には演算子 @ を使う

print("Product with * operator:")
print(a * a2)             # 演算子 * は要素同士の積
```

```
Inner product:
[[22 28]
 [49 64]]
Another Inner product:
[[22 28]
 [49 64]]
Inner product with @ operator:
[[22 28]
 [49 64]]
Product with * operator:
[[ 3  8 15]
 [24 35 48]]
```

`np.linalg` モジュールは標準的な行列分解、逆行列、固有値計算、連立一次方程式の解を計算する。
これらは、標準の線形代数ライブラリ BLAS, LAPACK が利用されている:

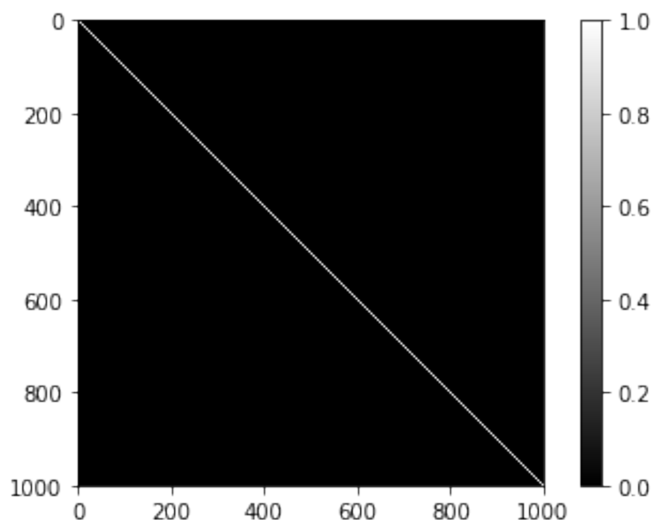
 CC BY-NC-ND

```
In [176]: import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv, qr

X = np.random.randn(1000,1000)
print("Sample:")
print(X)

print("M = X + X.T:")
M = X + X.T              # 対称行列を得る
print(M)
print("M^-1:")
print(inv(M))            # 逆行列を計算

print("M.dot(M^-1):")
print(M.dot(inv(M)))     # 単位行列になる

plt.imshow(M.dot(inv(M)), cmap=plt.cm.gray, vmin =0, vmax=1) # わからないので可視化してみる
plt.colorbar()
plt.show()

Q, R = qr(X)             # QR 分解  $X = Q \cdot R$
print(X - Q@R)
plt.imshow(X - Q@R, cmap=plt.cm.gray, vmin =0, vmax=1) # こっちも可視化してみる
plt.colorbar()
plt.show()
```

```
Sample:
[[ 0.40571503  0.15956574 -0.06914652 ... -0.95803849 -0.50062831
  -1.23623082]
 [-0.69119698  0.47263478 -1.53441367 ... -0.38879427  0.32756343
   0.27824722]
 [ 0.12305503 -0.9163054  -0.61096572 ...  1.41324628 -0.03782237
  -0.3678736 ]
 ...
 [ 0.31458231 -0.59750485  0.12898289 ...  1.17386313  1.23935184
  -0.80400266]
 [-0.4091114  -2.0871686  -0.1302501  ... -1.98855968  1.26733457
   0.62388348]
 [ 0.93818299 -0.14474953 -0.51140144 ...  1.02327206  1.9444703
   0.21271274]]
M = X + X.T:
[[ 0.81143007 -0.53163125  0.05390852 ... -0.64345619 -0.90973971
  -0.29804783]
 [-0.53163125  0.94526956 -2.45071907 ... -0.98629913 -1.75960517
   0.13349769]
 [ 0.05390852 -2.45071907 -1.22193145 ...  1.54222917 -0.16807247
  -0.87927504]
 ...
 [-0.64345619 -0.98629913  1.54222917 ...  2.34772626 -0.74920784
   0.2192694 ]
 [-0.90973971 -1.75960517 -0.16807247 ... -0.74920784  2.53466913
   2.56835378]
 [-0.29804783  0.13349769 -0.87927504 ...  0.2192694   2.56835378
```

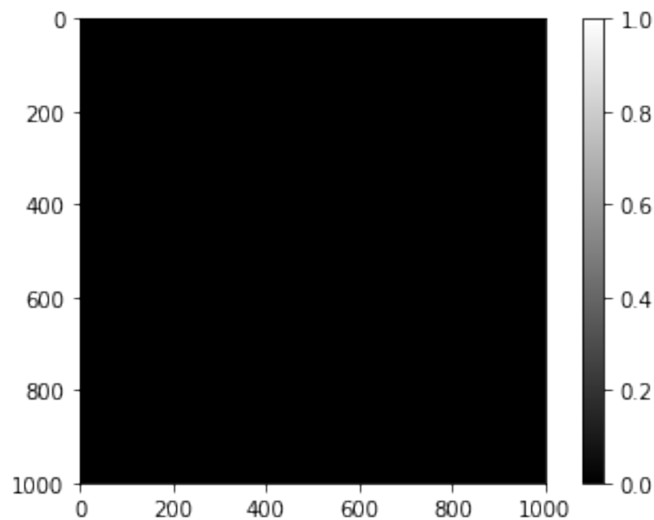
 CC BY-NC-ND

```
   0.42542549]]
M^-1:
[[-0.08117179 -0.00490618 -0.01493198 ...  0.01988024  0.06583788
   0.00511918]
 [-0.00490618 -0.01183699  0.00714379 ...  0.0056035   0.0209376
  -0.00281527]
 [-0.01493198  0.00714379 -0.00238892 ... -0.00885748  0.00474843
  -0.00124253]
 ...
 [ 0.01988024  0.0056035  -0.00885748 ... -0.00542105 -0.05005583
  -0.00439595]
 [ 0.06583788  0.0209376   0.00474843 ... -0.05005583 -0.13612463
  -0.01046945]
 [ 0.00511918 -0.00281527 -0.00124253 ... -0.00439595 -0.01046945
  -0.00940173]]
M.dot(M^-1):
[[ 1.00000000e+00  8.87484530e-15  2.03830008e-15 ... -2.82551760
e-14
   4.12239687e-14 -1.21430643e-14]
 [ 8.95117314e-15  1.00000000e+00  6.38725184e-15 ...  3.94823063
e-15
  -8.02136135e-15  1.50782165e-14]
 [ 1.46549439e-14  8.93729535e-15  1.00000000e+00 ... -8.60422844
e-15
  -3.08642001e-14  1.12826415e-14]
 ...
 [-3.59989816e-14  1.97064587e-14 -6.49480469e-15 ...  1.00000000
e+00
  -4.60742555e-15  1.52933222e-14]
 [-2.83661983e-14  5.06192310e-15 -5.91193761e-15 ...  3.10168558
e-15
   1.00000000e+00 -1.19071419e-14]
 [-4.21884749e-14 -1.79023463e-15  7.23032745e-15 ... -1.56541446
e-14
  -4.88498131e-15  1.00000000e+00]]
```



 CC BY-NC-ND

```
[[-6.10622664e-16 -1.33226763e-15 -4.16333634e-16 ... -4.44089210
e-16
   1.11022302e-16 -4.44089210e-16]
 [-1.11022302e-16 -1.49880108e-15  4.44089210e-16 ...  1.66533454
e-16
   2.77555756e-15  1.22124533e-15]
 [ 1.38777878e-17  2.22044605e-16  1.11022302e-16 ...  2.22044605
e-16
  -2.91433544e-16 -1.77635684e-15]
 ...
 [ 0.00000000e+00 -1.11022302e-16 -2.77555756e-17 ... -6.66133815
e-16
  -2.22044605e-16  6.66133815e-16]
 [ 0.00000000e+00  0.00000000e+00 -5.55111512e-17 ...  6.66133815
e-16
  -2.44249065e-15  1.44328993e-15]
 [ 0.00000000e+00 -2.77555756e-17 -1.11022302e-16 ... -6.66133815
e-16
  -8.88178420e-16 -5.82867088e-16]]
```

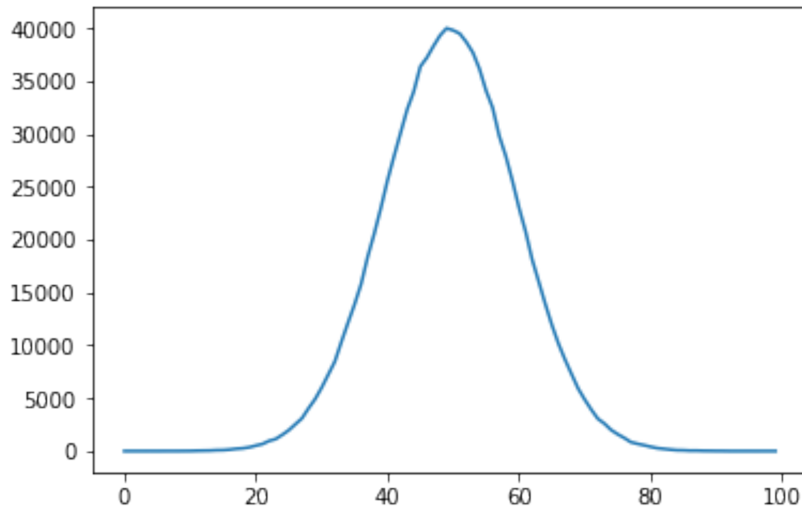


擬似乱数

`np.randome` モジュールは各種の確率分布をもつ乱数を生成する。
正規分布の生成には、`np.random.randn()` や `np.random.normal()` が使える:

In [24]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
np.random.poisson
N = 1000000
samples = np.random.normal(size=N)
bins = np.zeros(100)
for i in range(N):
        if -5 < samples[i] and samples[i] < 5:
            bins[int(samples[i] * 10 + 50)] += 1
plt.plot(bins)
```

Out[24]:
```
[<matplotlib.lines.Line2D at 0x111ec9518>]
```



このほか `np.random` では以下の乱数生成関数も使える:

分布	関数	備考
一様	rand()	
一様	uniform()	区間[0, 1)
正規	randn()	分散1, 平均0
正規	normel()	
二項	binomial()	

`np.random` 擬似乱数の seed は、`np.random.seed()` で与えることができる。
この seed の設定はプログラム内全ての np.random に影響する。
別の擬似乱数列が欲しい場合は以下のようにすれば良い:

In [210]:
```
rng = np.random.RandomState(1234)
rng.rand()
```

Out[210]:
```
0.1915194503788923
```

 CC BY-NC-ND