データ科学基礎演習B（5）

データ科学科目部会

モジュールの使い方

モジュールとは？

- 様々な機能（関数，クラス，etc.）をまとめて提供する仕組み（ライブラリ）
  - モジュールをインポートすることでプログラムを拡張
  - 様々なモジュールが提供・公開されているので，自分のやりたいことにあったモジュールを探してみよう

- モジュールのインポートの方法は4種類
  ① `import` モジュール名
  ② `import` モジュール名 `as` 別名
  ③ `from` モジュール名 `import` 機能
  ④ `from` モジュール名 `import` 機能 `as` 別名

モジュールのインポート

- モジュールの機能すべてをインポートする場合

`import` モジュール名

– モジュールと関数（クラス）をドット（.）でつなげて使う

モジュール ＋ ドット（.） ＋ 関数

※ mathモジュールのsqrt関数を使う場合

```
[1]   1   import math

[2]   1   math.sqrt(2)

1.4142135623730951
```

モジュールのインポート

- モジュールに別名を付けてインポートする場合

  `import` モジュール名 `as` 別名

  – モジュール名の代りに別名を使う

  別名 ＋ ドット(.) ＋ 関数

モジュール名が長い場合に便利

※ mathモジュールを別名「m」としてインポートする場合

```
[1]  1  import math as m

[2]  1  m.sqrt(2)

     1.4142135623730951
```

別名を指定すると元のモジュール名での呼び出し，math.sqrt(2)はエラーとなるので注意

モジュールのインポート

- 特定の機能のみをインポートする場合

  `from` モジュール名 `import` 機能

  – 機能を利用する際にモジュール名を省略できる

※ mathモジュールのsqrt関数を使う場合

```
[1]   1   from math import sqrt

[2]   1   sqrt(2)

      1.4142135623730951
```

モジュールのインポート

- 特定の機能に別名を付けてインポートする場合

```
from モジュール名 import 機能 as 別名
```

– 関数やクラス等を別名で使える（モジュール名不要）

※ mathモジュールのsqrtを別名「sq」としてインポート

```
[1]  1  from math import sqrt as sq

[2]  1  sq(2)

     1.4142135623730951
```

数値計算の入り口

numpyを使いこなそう

numpy モジュール

- ベクトルや行列などの計算を高速・効率的に行う機能を提供するライブラリ
- numpyのインポートとバージョンの調べ方
  - numpyには別名「np」を使うことが多い
    - 以降の説明ではnpを使う
  - バージョンを知りたい時は __version__ を調べる

```
[1]  1  import numpy as np

[2]  1  np.__version__

     '1.19.5'
```

numpy 配列の使い方

- numpyの配列はndarray（別名array）
  - 行列のように加減算等ができる
- ndarrayの使い方
  - 中身を指定して1次元配列を作成

```
a = np.array([1, 2, 3, 4])
```

  - 中身を指定して2次元配列を作成

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

0行目　　1行目

1	2	3	4
5	6	7	8

numpy を使った等差数列の作り方

- np.arange関数を使った等差数列の作成方法は3種類

  例）1，1.5，2.0，2.5，… のような配列

① np.arange(終了値)
② np.arange(開始値，終了値)
③ np.arange(開始値，終了値，ステップ幅)

np.arange(終了値)

- [0,1,2,…,終了値-1] の差が1の等差数列
  - 配列の最初の値は 0
  - 配列の最後の値は 終了値-1
  - 配列の要素数 ＝ 終了値
  - np.array(range(終了値)) と同じ結果が得られる

```
[2]   1  np.arange(5)

      array([0, 1, 2, 3, 4])

[3]   1  np.array(range(5))

      array([0, 1, 2, 3, 4])
```

np.arange(開始値，終了値)

- [開始値,開始値+1,…,終了値-1] を返す
  - 配列の最初の値は 開始値
  - 配列の最後の値は 終了値-1
  - 要素間の差は 1

```
[2]   1  np.arange(1,5)

      array([1, 2, 3, 4])

[3]   1  np.array(range(1,5))

      array([1, 2, 3, 4])
```

np.arange(開始値，終了値，ステップ幅)

- [開始値,開始値+ステップ幅,…,終了値-ステップ幅]
  - 配列の最初の値は 開始値
  - 配列の最後の値は 終了値ーステップ幅
  - 配列の要素の間隔は ステップ幅(小数を指定可)

```
[2]  1  np.arange(1,2,0.1)

     array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])

[3]  1  np.arange(1,2,0.3)

     array([1. , 1.3, 1.6, 1.9])
```

numpy 配列のサイズ

- 全要素数 → size（3×3配列の場合は9）
- 各次元のサイズ → shape
  - shapeは各次元の要素数をタプルで返す

```
[1]  1  import numpy as np

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

(2, 4)

     1  a

array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

行数　列数

指定したサイズの多次元配列を作成

- 単位行列の作り方

```
a = np.eye(3) #3×3の単位行列
```

- ゼロ行列の作り方

```
a = np.zeros((2,4)) #2×4のゼロ行列
```

- 全要素が1の多次元配列の作り方

```
a = np.ones((3,4,5)) #3×4×5の配列(全要素1)
```

サイズをタプルで指定
（shapeと同じ）

numpy 配列の要素アクセス(1)

- 要素へのアクセス(インデックスは 0 はじまり)
  - N行M列の行列(2次元配列)の場合のインデックス
    - 行: 0 ～ N-1
    - 列: 0 ～ M-1
  - 1行1列目の要素にアクセス

    a[1,1]

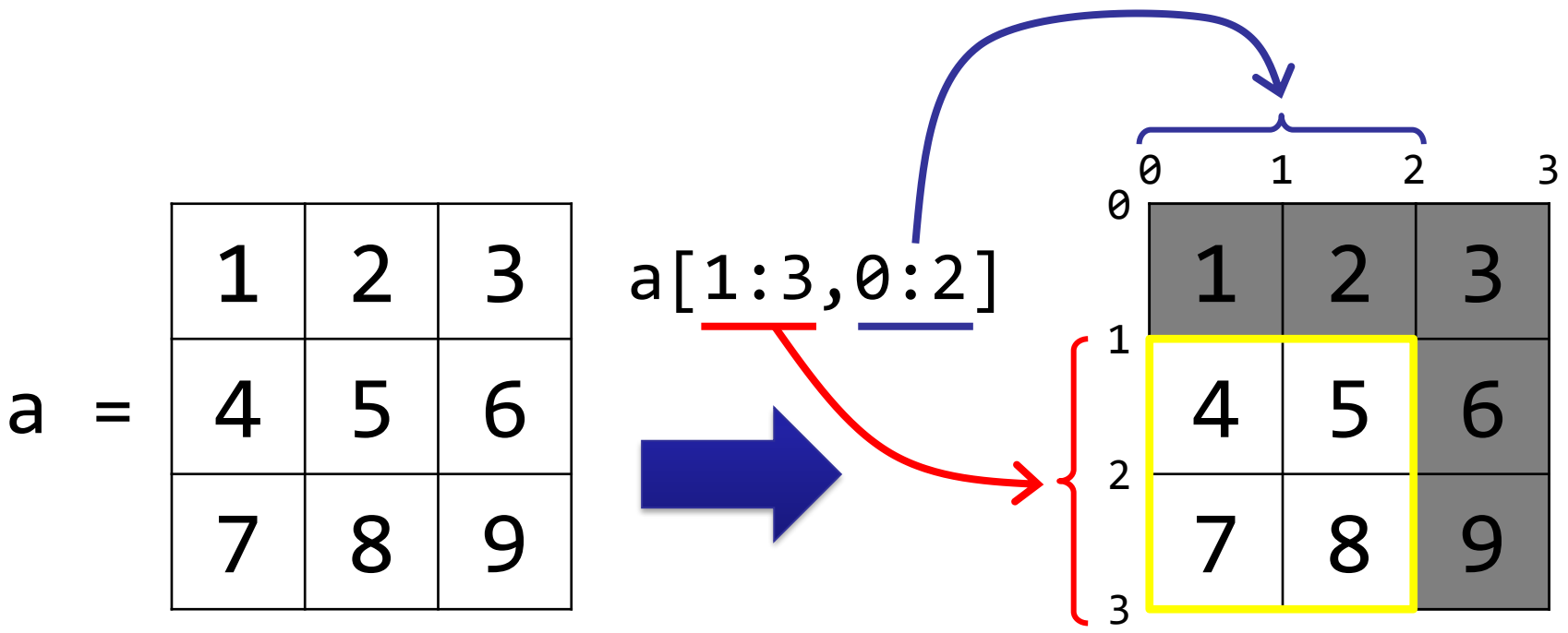
```
[1]  1  import numpy as np
     2  a = np.array([[1,2,3,4],[5,6,7,8]])

[2]  1  a

array([[1, 2, 3, 4],
       [5, 6, 7, 8]])

[3]  1  a[1,1]

6

[4]  1  a[1,3]

8
```

numpy 配列の要素アクセス(2)

- スライスを使うと部分配列が取得できる

```
配列の変数[start1:stop1, start2:stop2,…]
```

1次元目の範囲　　2次元目の範囲



※startを省略すると先頭，stopを省略すると最後が指定される

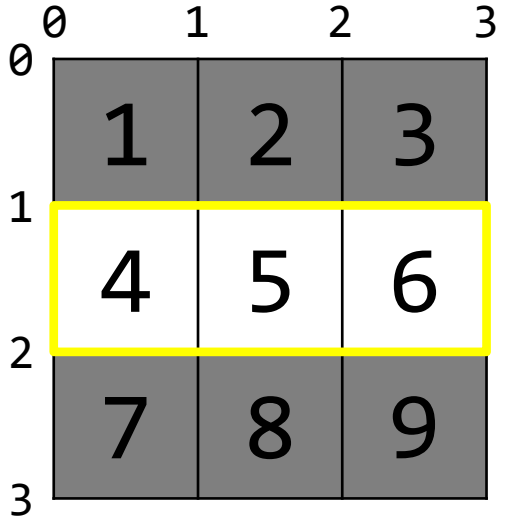
numpy 配列の要素アクセス(3)

- スライスを使って特定の行もしくは列を抽出

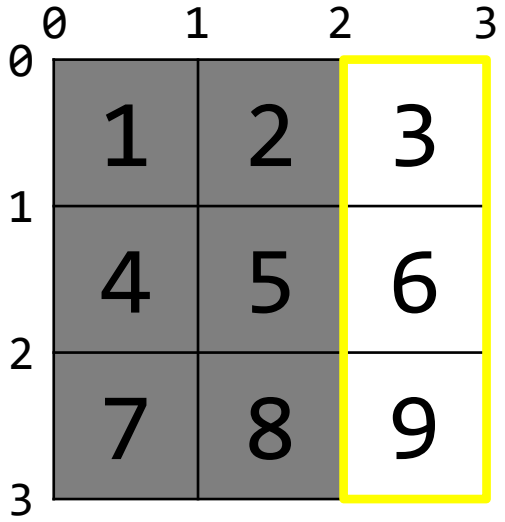
a =

1	2	3
4	5	6
7	8	9

a[1,:]



a[:,2]



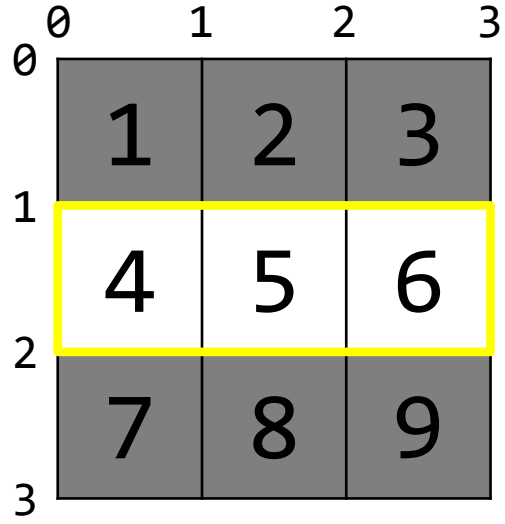
numpy 配列の要素アクセス(3)

- スライスを使って特定の行もしくは列を抽出

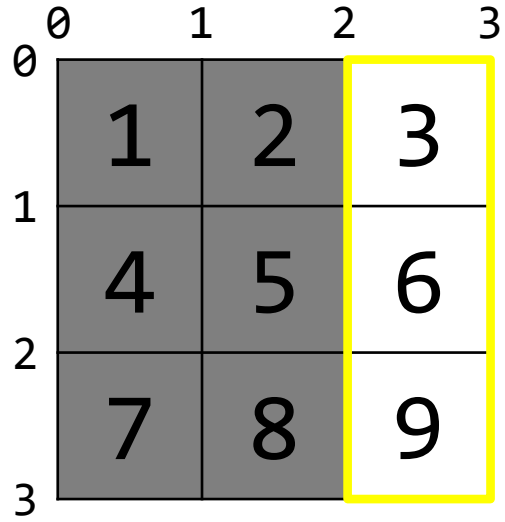
a =

1	2	3
4	5	6
7	8	9

a[1,:]



a[:,2]



このスライス指定では
抽出結果はいずれも
行ベクトル
になる

numpy 配列の要素アクセス(3)

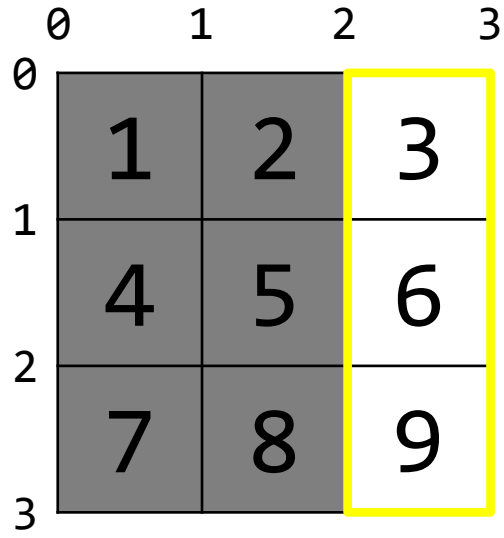
- スライスを使ってある列を列ベクトルとして抽出

a =

1	2	3
4	5	6
7	8	9

a[:,2:3]

列の範囲を明記



numpy 配列の要素アクセス(3)

- スライスを使ってある列を列ベクトルとして抽出

```
[2]  1  import numpy as np
     2  a = np.arange(1,10).reshape(3,3)
     3  a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
[3]  1  a[:,1]
```

```
array([2, 5, 8])
```

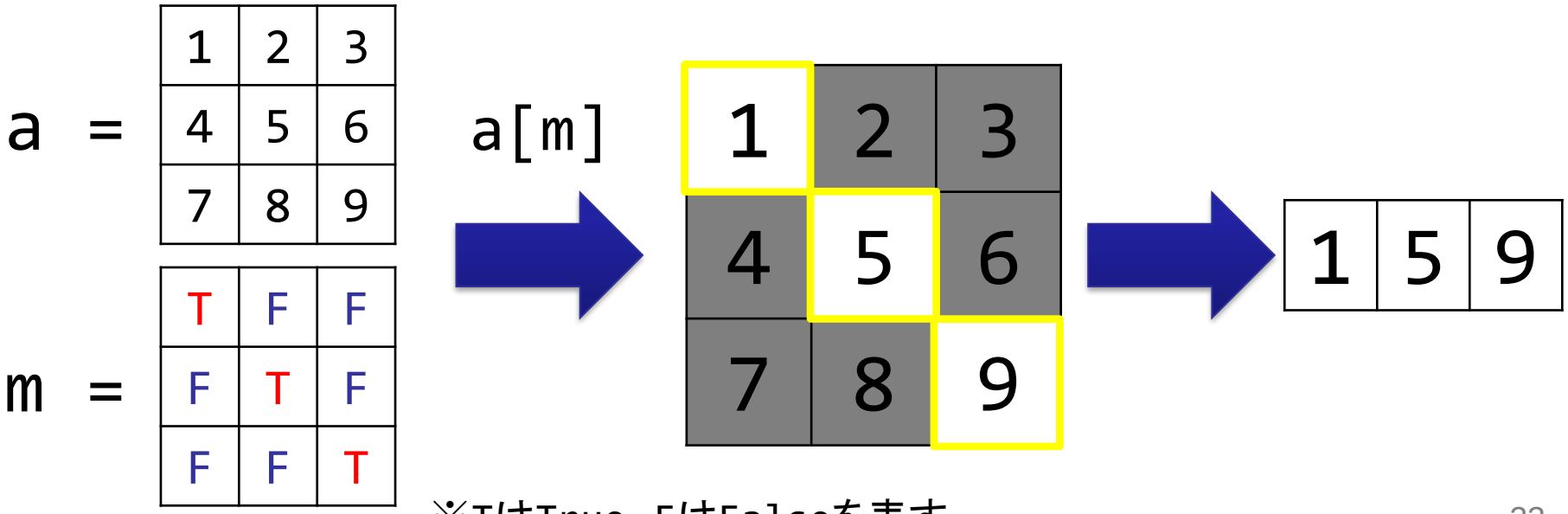
```
[4]  1  a[:,1:2]
```

```
array([[2],
       [5],
       [8]])
```

numpy 配列の要素アクセス(4)

- ブール配列をマスクとして特定の要素を抽出
  - マスクの各要素は True もしくは False
  - マスクの配列サイズ ＝ 抽出対象配列のサイズ

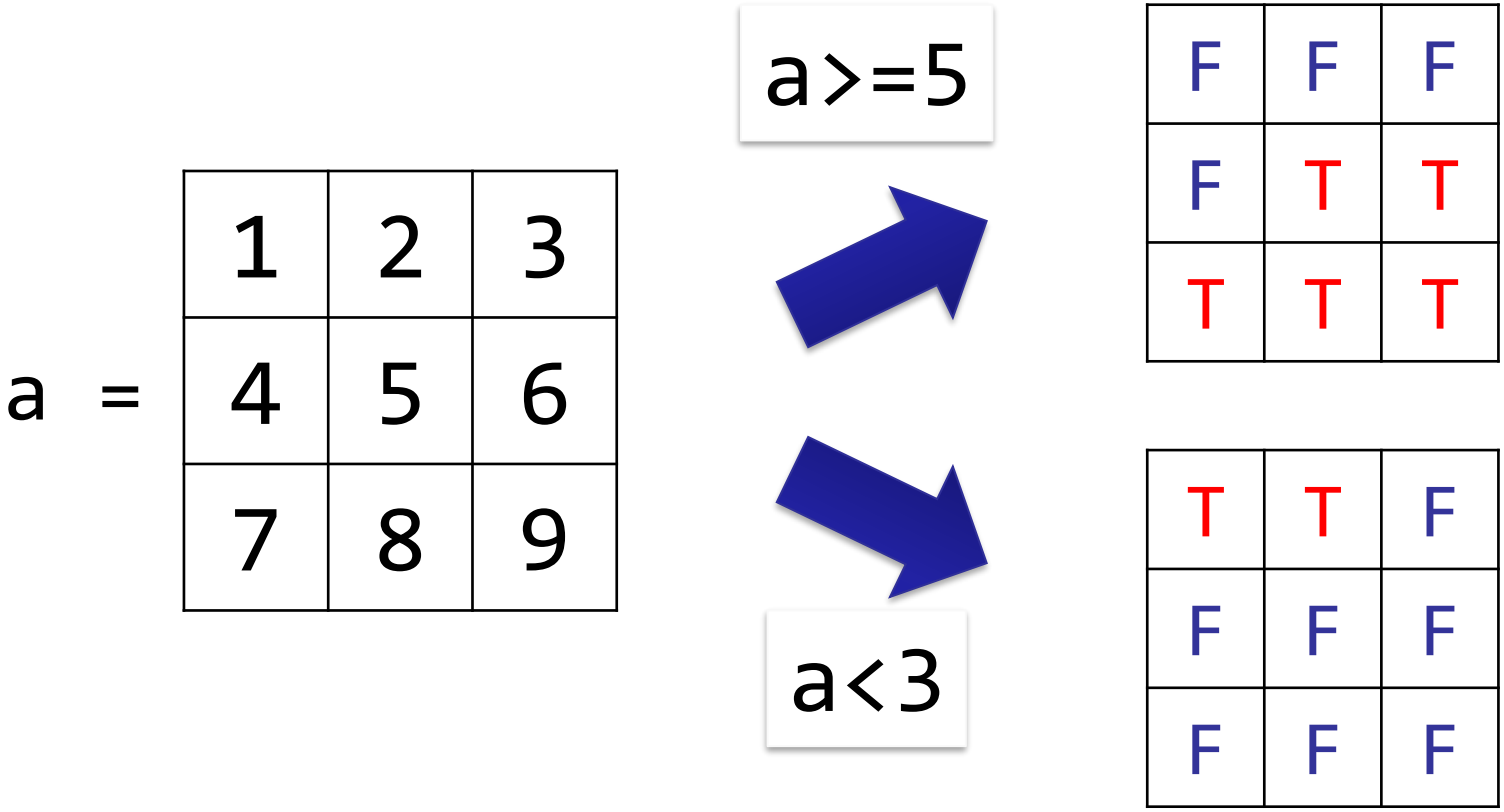
配列の変数[マスク]



※TはTrue，FはFalseを表す

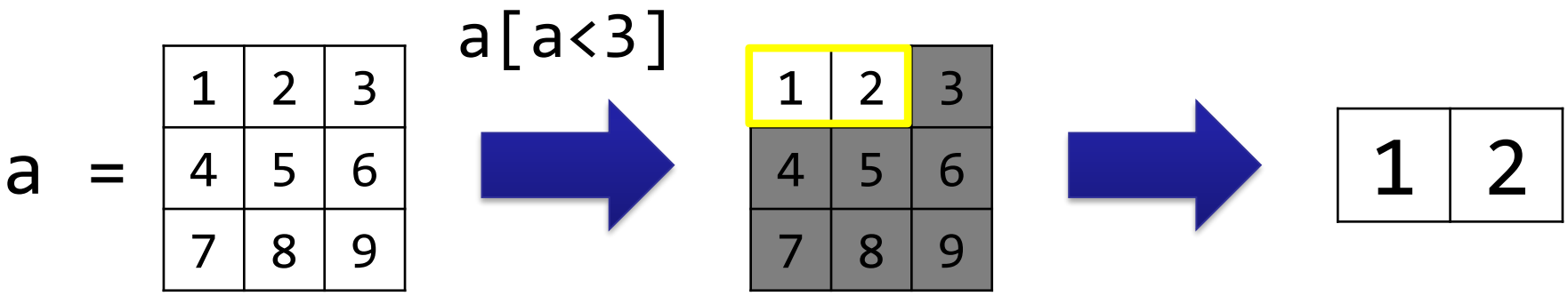
numpy 配列の要素アクセス(5)

- 比較演算子を使ってマスク(ブール配列)を生成
  - 値が5以上の所のみTrue → a>=5
  - 値が3未満の所のみTrue → a<3



numpy 配列の要素アクセス(5)

- 比較演算子による特定要素の抽出




```
[1]  1  import numpy as np
     2  a = np.array([1,2,3,4,5,6,7,8,9]).reshape((3,3))

[2]  1  a<3

array([[ True,  True, False],
       [False, False, False],
       [False, False, False]])

[3]  1  a[a<3]

array([1, 2])
```

numpy 配列のサイズ変更（reshape）

- 要素数を変えずに配列の形状を変える方法
  - 形状を変更するのみで要素の値は保持される
    - 例1）３×4の行列を2×6の行列に変更
    - 例2）３×3の行列を9×1のベクトルに変更

配列の変数.reshape((変更後のサイズ))

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

[2]  1  a

array([1, 2, 3, 4, 5, 6, 7, 8, 9])

[3]  1  a.reshape((3,3))

array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

numpy 配列の四則演算

- 配列同士の四則演算(+, -, *, /, 等々)
  - 要素単位で四則演算をした結果が得られる

```
[1]  1  import numpy as np
     2  a = np.array([1,2,3,4,5,6,7,8,9]).reshape((3,3))
```

```
[2]  1  a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
[3]  1  a+a
```

```
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

```
[4]  1  a*2
```

```
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

numpy の数学関数

- スカラー／多次元配列（ベクトル，行列，etc.）に対するさまざまな数学関数が利用可能

  ※https://numpy.org/doc/stable/reference/routines.math.html

  - よく使う関数の例
    - ceil(x)　　# x以上の最小の整数
    - floor(x)　　# x以下の最大の整数
    - fabs(x)　　# 絶対値
    - exp(x)　　# e^x
    - power(x,y)　# x^y
    - sqrt(x)　　# $\sqrt{x}$
    - log(x)　　# 対数
    - sin(x), cos(x), tan(x)　# xの単位はラジアン

numpy の数学関数

- スカラー／多次元配列（ベクトル，行列，etc.）に対するさまざまな数学関数が利用可能

  ※https://numpy.org/doc/stable/reference/routines.math.html

  - 数学で使われる定数
    - pi　　# π
    - e　　# 自然対数の底
    - inf　　# 無限大
    - nan　　# 非数（not a number）

```
[1]  1  import numpy as np
     2  a = np.array([1.5, 2.0, 3.5])

[2]  1  np.power(2, 2)

     4

[3]  1  np.power(a, 2)

     array([ 2.25,  4.  , 12.25])

[4]  1  np.ceil(a)

     array([2., 2., 4.])

[5]  1  np.pi

     3.141592653589793
```

numpy 配列の和・平均・最大・最小（1）

- 配列内の要素の総和

  ```
  a.sum()
  ```

- 配列内の要素の平均

  ```
  a.mean()
  ```

- 配列内の要素の最大

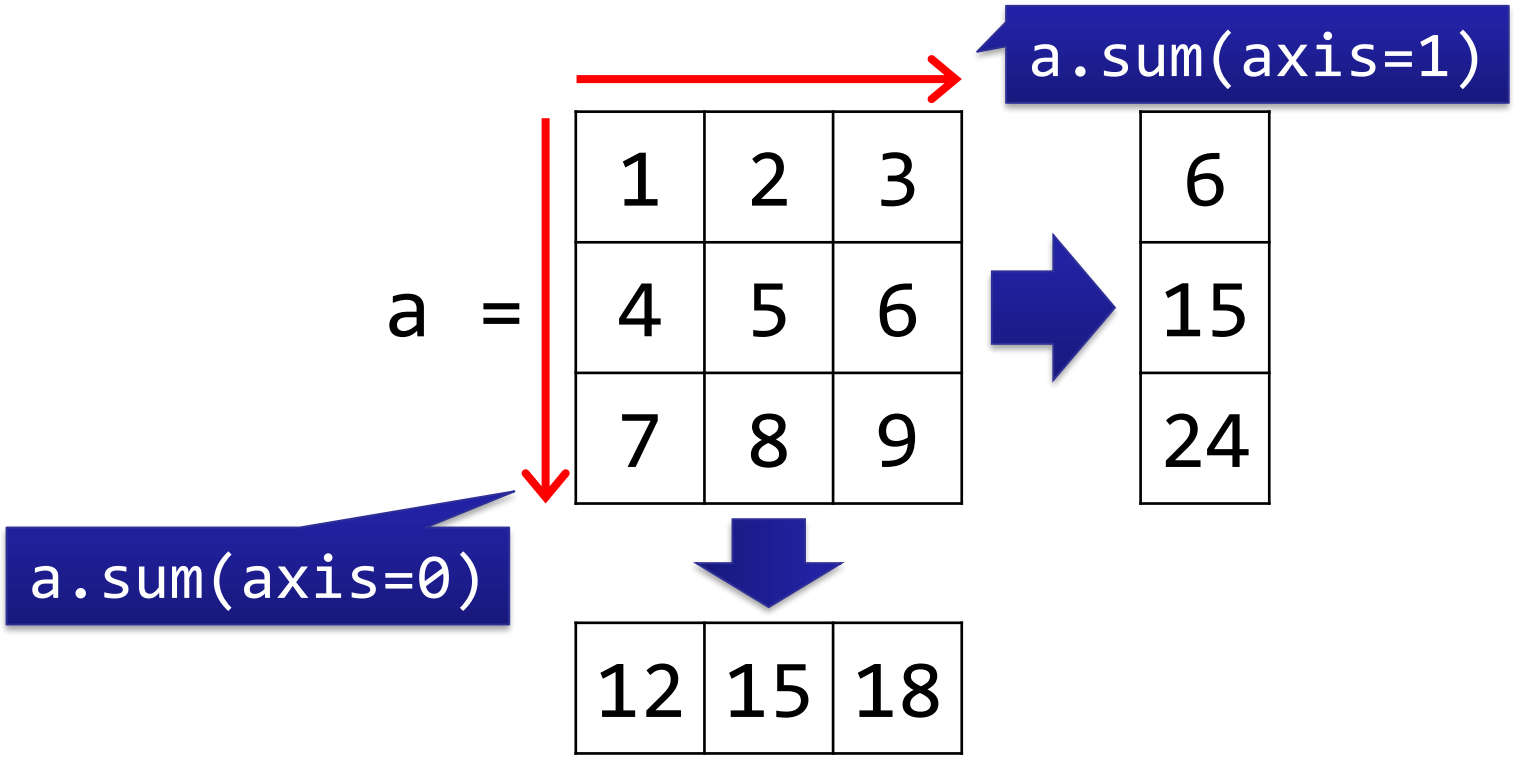
  ```
  a.max()
  ```

- 配列内の要素の最小

  ```
  a.min()
  ```

numpy 配列の和・平均・最大・最小(2)

- axis引数を指定することで特定軸方向の総和・平均・最大・最小を求めることができる
  - axis=0： 行インデックスが増える方向(**縦**方向)に値を操作
  - axis=1： 列インデックスが増える方向(**横**方向)に値を操作



numpy 配列（ndarray）の使い方練習

① 3×3の単位行列を作成してみよう

② ①の配列に×10 をしてみよう

③ 3×3の行列

1	2	3
4	5	6
7	8	9

を ② に掛けてみよう

④ ③の配列の右上の2×2の部分行列をスライスを使って切り出してみよう

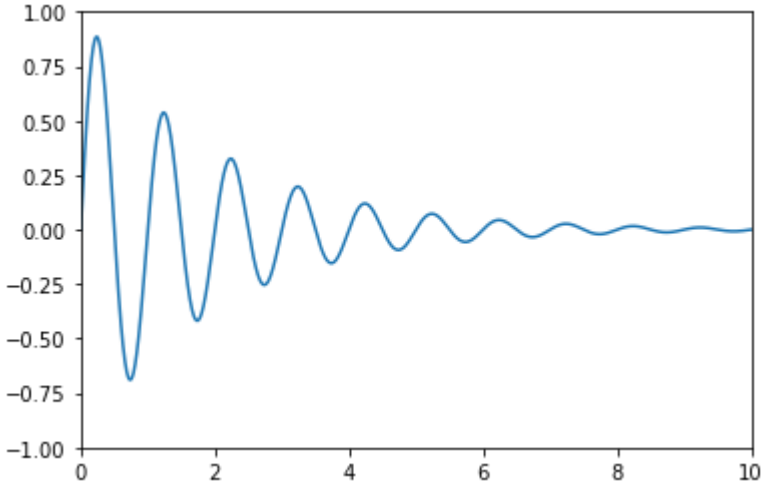
⑤ ④の部分行列を4×1のベクトルにサイズ変更してみよう

データの可視化

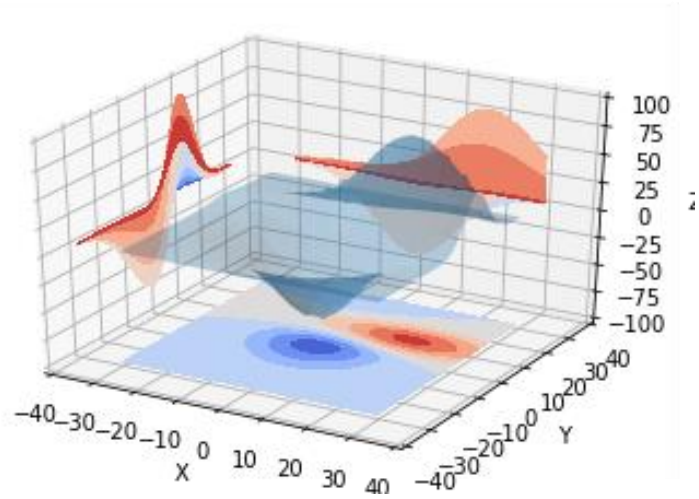
matplotlib を使いこなそう

matplotlib とは

- 2D／3Dのグラフを描画するためのライブラリ
  - 折れ線グラフ
  - 散布図
  - 棒グラフ
  - 円グラフ
  - etc.



- プログラムの解析結果を可視化する際に利用



matplotlib の使い方

- matplotlibモジュールのpyplotをインポート

グラフをページ内に表示するためのおまじない

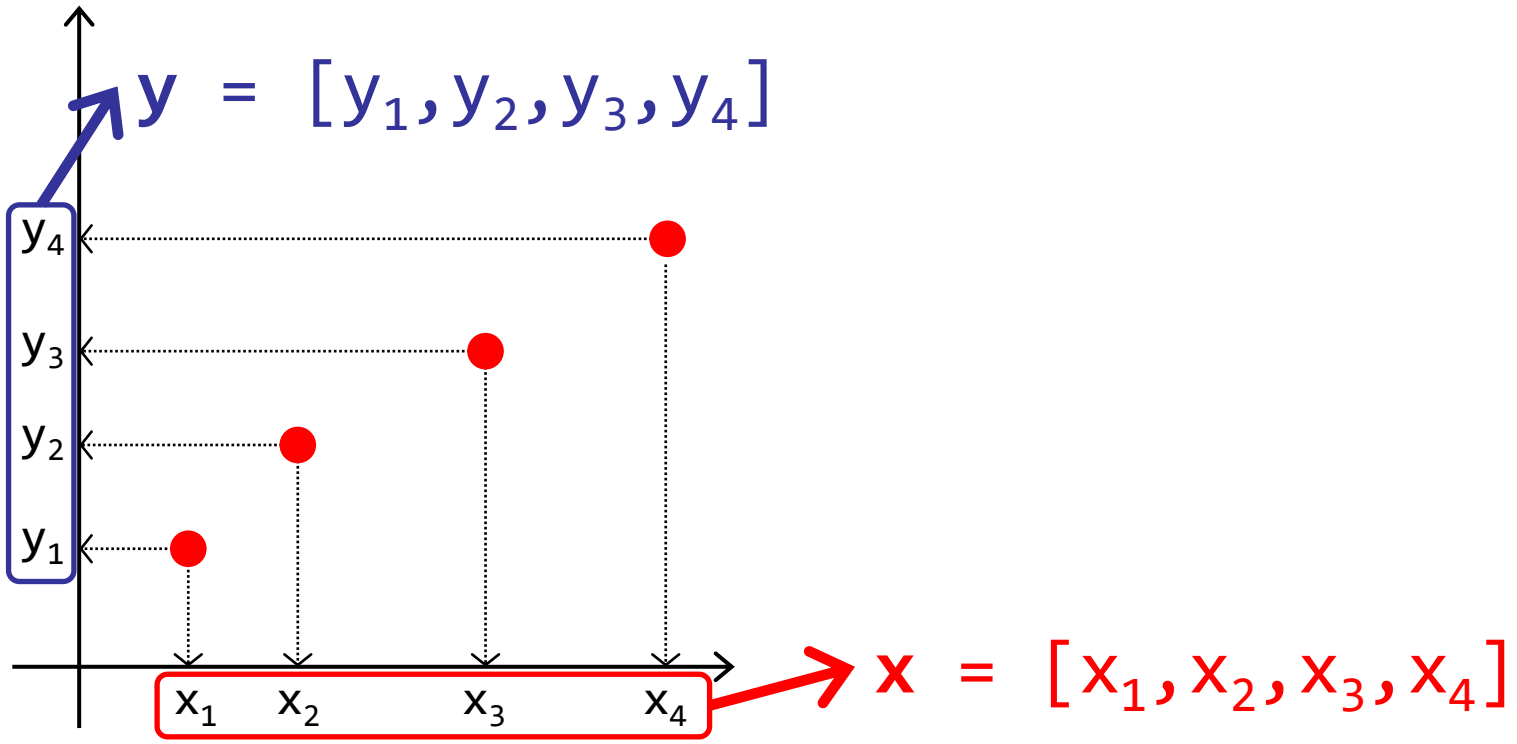
この2つは同じ意味

```
%matplotlib inline
#from matplotlib import pyplot as plt
import matplotlib.pyplot as plt
```

matplotlibは名前が長いため別名を付けるのが便利（以降は plt を使って説明）

matplotlib の使い方（2Dグラフの描画）

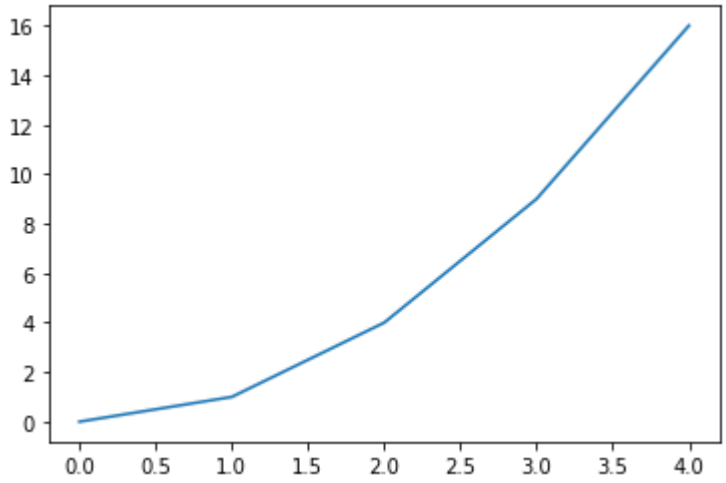
- X軸とY軸の値をそれぞれ格納したベクトルを準備
  - X軸の値を格納したベクトル → **x**
  - Y軸の値を格納したベクトル → **y**



matplotlib の使い方（2Dグラフの描画）

- X軸とY軸の値をそれぞれ格納したベクトルを準備
  - X軸の値を格納したベクトル → `x = np.arange(5)`
  - Y軸の値を格納したベクトル → `y = x*x`
- `plt.plot`関数に x と y を渡してグラフを描画
  - `plt.plot(x,y)`

```
[1]  1  %matplotlib inline
     2  import matplotlib.pyplot as plt
     3  import numpy as np

[2]  1  x = np.arange(5)
     2  y = x*x

     1  plt.plot(x,y)
```

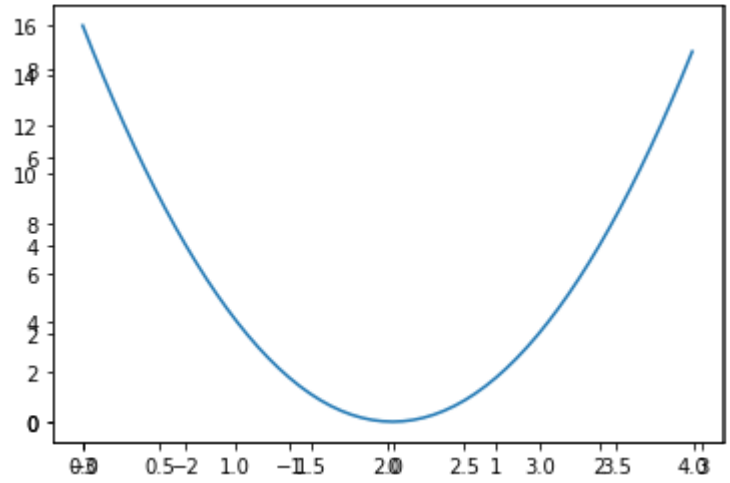


matplotlib の使い方（2Dグラフの描画）

- グラフをより細かく描画したい場合は？
  - X軸の値を格納したベクトルの範囲を調整

    `x = np.arange(5)` → `x = np.arange(-3,3,0.1)`
  - Y軸の値をxを使って再計算 → `y = x*x`
- plt.plot関数に x と y を渡してグラフを描画

```
[1]  1  %matplotlib inline
     2  import matplotlib.pyplot as plt
     3  import numpy as np

[2]  1  x = np.arange(-3, 3, 0.1)
     2  y = x*x

     1  plt.plot(x, y)
```

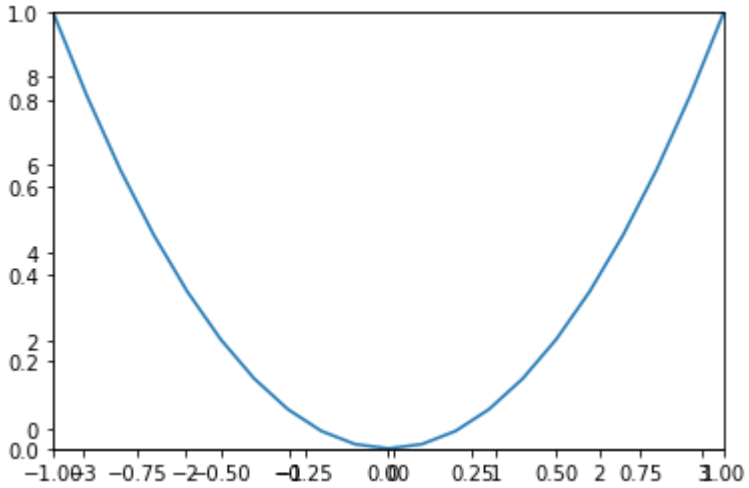


matplotlib の使い方（表示範囲の調整）

- X軸の表示範囲をグラフ描画後に調整する場合は？
  - X軸の表示範囲を[-1,1]にする場合 → `plt.xlim(-1,1)`
  - Y軸の表示範囲を[0,1]にする場合 → `plt.ylim(0,1)`

`plt.plot`関数と同じコードブロックに書くこと！！

```
[1]  1  %matplotlib inline
     2  import matplotlib.pyplot as plt
     3  import numpy as np

[2]  1  x = np.arange(-3,3,0.1)
     2  y = x*x

     1  plt.xlim(-1,1)
     2  plt.ylim(0,1)
     3  plt.plot(x,y)
```



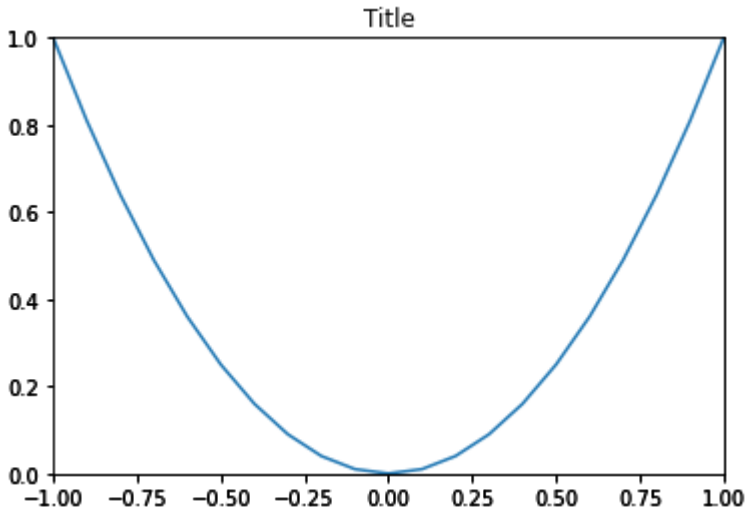
matplotlib の使い方（グラフのタイトル）

- グラフにタイトルを付けるには？
  - plt.title関数にタイトルを渡す

```
plt.title('Title')   #日本語はうまく表示できない
```

plt.plot関数と同じコードブロックに書くこと！！

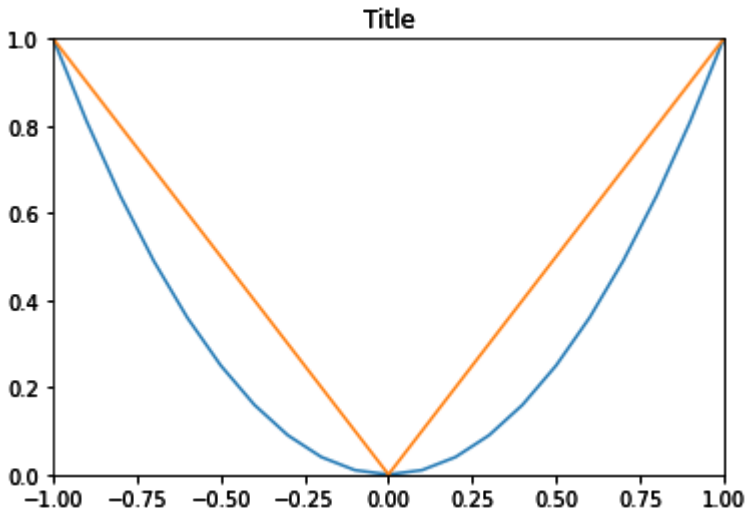
```
[1]  1  %matplotlib inline
     2  import matplotlib.pyplot as plt
     3  import numpy as np

[2]  1  x = np.arange(-3,3,0.1)
     2  y = x*x

     1  plt.xlim(-1,1)
     2  plt.ylim(0,1)
     3  plt.title('Title')
     4  plt.plot(x,y)
```



matplotlib の使い方（2Dグラフの描画）

- 別のグラフを重ねて表示する場合は？
  - 同じコードブロック内で2回 `plt.plot` を実行する

```
[1]  1  %matplotlib inline
     2  import matplotlib.pyplot as plt
     3  import numpy as np

[2]  1  x = np.arange(-3,3,0.1)
     2  y = x*x
     3  y2 = np.fabs(x)

     1  plt.xlim(-1,1)
     2  plt.ylim(0,1)
     3  plt.title('Title')
     4  plt.plot(x,y)
     5  plt.plot(x,y2)
```



【補足】matplotlib の発展的な使い方

- 複雑なグラフの描画も挑戦したい場合は下記キーワードを調べてみてください
  - 軸にラベルを付ける → `plt.xlabel(), plt.ylabel()`
  - グラフに罫線を付ける → `plt.grid()`
  - グラフを複数並べる → `plt.subplot()`

- より細かくグラフの描画を制御したい場合は下記キーワードを調べてみてください
  - `matplotlib + オブジェクト指向`

課題

- numpyとmatplotlibの操作をマスターしよう

- 下記URLから演習課題のノートブックを自身のGoogle Driveにコピーし，Google Collaboratoryを起動して各設問に回答すること
  – https://bit.ly/3qMRbvw

- 演習課題を提出する際は，ノートブックのURLで共有し，そのURLを提出すること

課題提出時の注意点

- 課題提出の際には提出前に必ず以下2点を確認すること
  - Google Collaboratoryの共有設定の際には『リンクを知っている全員』にチェックを入れる
  - 課題提出時に貼り付けたURLの末尾が「?usp=sharing」となっている（共有設定を開き、「リンクのコピー」ボタンを押して共有用のURLをコピー＆ペーストする）