



# Python for bioinformatics



どんぐり研究所 孫 建強

Contents in this document are licensed under [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/).

# コンテンツ

## Introduction

1. プログラミング言語概要
2. Python 環境構築

## Basic

3. データ型
4. 基本文法

## Packages

5. テキスト処理
6. 数値計算 NumPy
7. データ処理 Pandas
8. データ可視化 matplotlib
9. バイオインフォマティクス

## Advanced

10. オブジェクト指向

# 数値計算 NumPy



- 1 次配列
- 2 次配列
- 3 次配列
- データ読み取り



アプリ開発やシステム管理など様々な用途で利用できるような汎用機能を提供する。

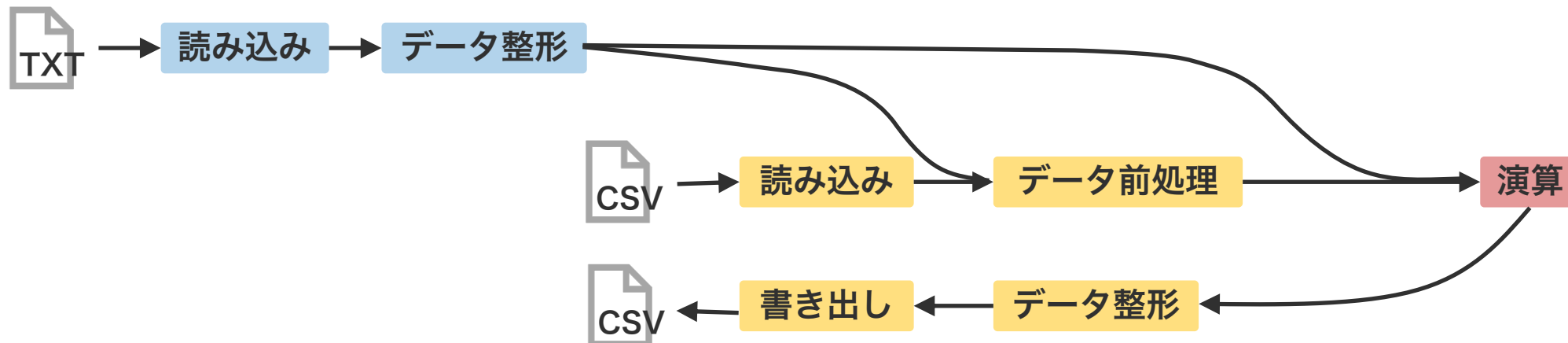
- リスト
- 2次元リスト

CSV ファイルの読み書きや表データの操作や整形などに特化した機能を提供する。

- シリーズ
- データフレーム

整形されたデータに対して、情報量をできるだけ落とさずに、高速に演算を行う機能を提供する。

- 配列
- 2次元配列



# 数値計算



- 1 次配列
- 2 次配列
- 3 次配列
- データ読み取り

# 1 次元配列 / np.array

NumPy の 1 次元配列は、Python の 1 次元リストを拡張したようなオブジェクトである。NumPy 配列は、NumPy で用意された関数を使用して作成する。例えば、`np.array` 関数を使えば、任意の Python のリストを配列に変換できる。また、`np.full` や `np.linspace` などの関数を使えば、すべての要素が同じ値であるような配列や等差数列のような配列を作することもできる。

`np.array`

`np.zeros`

`np.ones`

`np.full`

`np.arange`

`np.linspace`

```
import numpy as np
```

```
a = [1, 1, 2, 3, 5, 8]
```

```
a
```

```
# [1, 1, 2, 3, 5, 8]
```

```
b = np.array(a)
```

```
b
```

```
# array([1, 1, 2, 3, 5, 8])
```

```
c = np.array([1, 1, 2, 3, 5, 8])
```

```
c
```

```
# array([1, 1, 2, 3, 5, 8])
```

# 1 次元配列 / np.full

すべての要素が同じ値であるような配列を作るとき、`np.zeros`, `np.ones`, `np.full` 関数を使用すると便利である。`np.zeros` と `np.ones` 関数は、関数名の通り、すべての要素が 0 または 1 であるような配列を作るときに使用する。また、また、`np.full` 関数は、任意の値で初期化された配列を作るときに使用する。

`np.array`

`np.zeros`

`np.ones`

`np.full`

`np.arange`

`np.linspace`

```
import numpy as np
```

```
a = np.zeros(5)
```

```
a
```

```
# array([0, 0, 0, 0, 0])
```

```
b = np.ones(8)
```

```
b
```

```
# array([1, 1, 1, 1, 1, 1, 1, 1])
```

```
c = np.full(5, np.nan)
```

```
c
```

```
# array([nan, nan, nan, nan, nan])
```

# 1 次元配列 / np.arange

等差数列からなる配列を作るとき、np.arange 関数を使用する。np.arange 関数は、start (数列の最初の値), stop (数列の範囲), step (間隔) の 3 つの引数を受け取り、それらに基づいて等差数列からなる配列を作る。start を省略すると start=0 となり、step を省略すると step=1 となる。

np.array

np.zeros

np.ones

np.full

np.arange

np.linspace



関数名に注意。np.arrange ではない。NumPy 前身であった Numeric 時代で使われた arange 関数の省略形として arange が使われていたことに由来する。

```
import numpy as np
```

```
a = np.arange(5)
```

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

```
b = np.arange(1, 6)
```

```
b  
# array([1, 2, 3, 4, 5])
```

```
c = np.arange(1, 6, 2)
```

```
c  
# array([1, 3, 5])
```

```
d = np.arange(10, 0, -2)
```

```
d  
# array([10, 8, 6, 4, 2])
```



# 1 次元配列 / np.linspace

等差数列からなる配列を作るとき、`np.linspace` 関数も利用できる。この関数は、`start` (数列の最初の値), `stop` (数列の最後の値), `num` (要素数) の 3 つの引数を受け取り、それらに基づいて等差数列からなる配列を作る。`num` を省略すると `num=50` となる。

`np.array`

`np.zeros`

`np.ones`

`np.full`

`np.arange`

`np.linspace`



関数名に注意。`np.linspace` ではない。`Linearly space vectors` を生成する関数のため、`linspace` である。

```
import numpy as np
```

```
a = np.linspace(1, 9, 3)
a
# array([1., 5., 9.]
```

```
b = np.linspace(1, 9, 5)
b
# array([1., 3., 5., 7., 9.]
```

```
c = np.linspace(1, 0, 5)
c
# array([1.   , 0.75, 0.5  , 0.25, 0.   ])
```

# 1 次元配列 / 数値計算

NumPy には、切り上げや切り捨てを行う `np.ceil` や `np.floor`、対数化を行う `np.log`、三角関数の値を計算する `np.sin`, `np.cos` や `np.tan` など、平均や分散を計算する `np.mean` や `np.std` など、多くの関数が用意されている。これらの関数名をすべて覚える必要はなく、使いたいときにその使い方を調べればよい。

```
import numpy as np

a = np.array([1, 5, 10, 50, 100])

np.log10(a)
# array([0., 0.69897, 1., 1.69897, 2.])

np.sqrt(a)
# array([1., 2.2360, 3.1622, 7.0710, 10.])

np.mean(a)
# 33.2

np.std(a)
# 37.722142038860945
```

# 1 次元配列

NumPy の 1 次元配列は、リストほぼ同じように取り扱うことができる。配列は基本的に全要素を束ねて使うことが多いが、一部の要素だけを取り出したり、変更したりすることもできる。その際、配列の先頭から数えて何番目の要素を取り出したいのかを、整数で指定する必要がある。

```
import numpy as np

w = np.array([12, 10, 11, 13, 11])


w[0]
# 12

w[1]
# 10

w[2]
# 11

w[2] = 9

w
# array([12, 10, 9, 13, 11])
```



# 1 次元配列 / スライス

Python のリストと同様に、隣り合う要素をスライスして取得することもできる。スライスを行うとき、スライスの開始位置と終了（手前の）位置をコロン（:）で区切って指定する。

スライスにより取り出された部分配列は、元の配列への参照となっている。したがって、スライスで得られた部分配列の値を変更すると、元の配列の値も変化してしまう。

```
a = np.array([1, 3, 5, 7, 9,
              2, 4, 6, 8, 0])
```

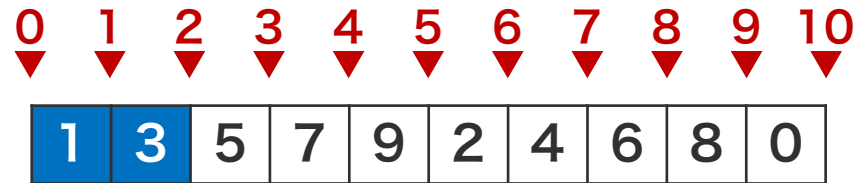
```
b = a[3:6]
b
# np.array([7, 9, 2])
```

```
b[0] = 0
b[1] = 0
b[2] = 0
```

```
b
# np.array([0, 0, 0])
```

```
a
# np.array([1, 3, 5, 0, 0,
#           0, 4, 6, 8, 0])
```

# 1 次元配列 / スライス



```
a = np.array([1, 3, 5, 7, 9,  
             2, 4, 6, 8, 0])
```

```
a[0:2]  
# np.array([1, 3])
```

```
a[2:6]  
# np.array([5, 7, 9, 2])
```

```
a[:4]  
# np.array([1, 3, 5, 7])
```

```
a[5:]  
# np.array([2, 4, 6, 8, 0])
```

# 1 次元配列 / スライス

スライスを行うとき、開始位置と終了位置の他に、ステップ数を与えることもできる。1 要素おきに値を 1 つ取り出す場合などに便利である。また、配列では、連続していない要素を同時に取り出すこともできる。

```
import numpy as np

a = np.array([1, 3, 5, 7, 9,
              2, 4, 6, 8, 0])

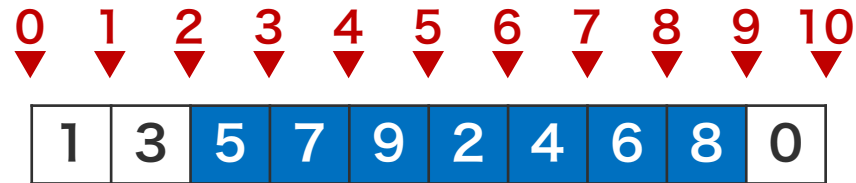
a[2:9]

a[2:9:1]

a[2:9:3]

a[[0, 2, 4, 6]]
```

# 1 次元配列 / スライス



```
import numpy as np
```

```
a = np.array([1, 3, 5, 7, 9,  
             2, 4, 6, 8, 0])
```

```
a[2:9]  
# array([5, 7, 9, 2, 4, 6, 8])
```

```
a[2:9:1]  
# array([5, 7, 9, 2, 4, 6, 8])
```

```
a[2:9:3]  
# array([5, 2, 8])
```

```
a[[0, 2, 4, 6]]  
# array([1, 5, 9, 4])
```

# 1 次元配列 / フィルター

配列から要素を取り出すとき、位置番号で指定するほか、True または False からなる配列（ブーリアンベクトル）で指定することもできる。このとき、ブーリアンベクトルの長さは、操作対象となる配列の長さと同じでなければならない。

```
import numpy as np

a = np.array([ 2, 4, 6, 8])
k = np.array([True, True, False, True])
```

```
a[k]
```

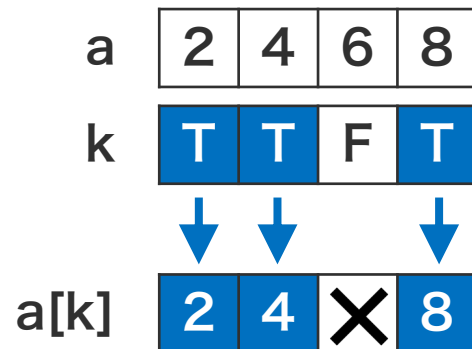
```
a = np.array([ 2, 4, 6, 8])
k = np.array([False, True, True, True])
```

```
a[k]
```



# 1 次元配列 / フィルター

フィルター `k` を用いて、ベクトル `a` の要素をフィルタリングしているイメージ。



```
import numpy as np
```

```
a = np.array([ 2, 4, 6, 8])
```

```
k = np.array([True, True, False, True])
```

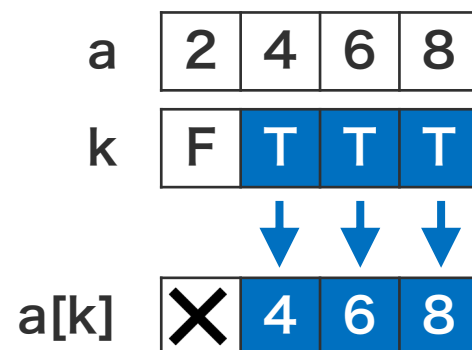
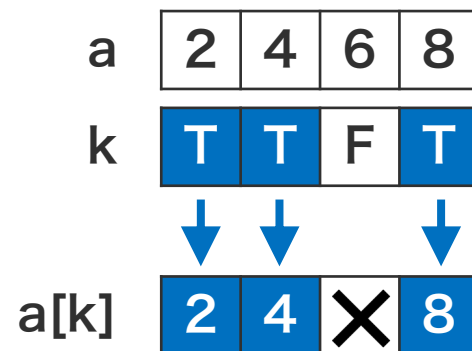
```
a[k]  
# array([2, 4, 8])
```

```
a = np.array([ 2, 4, 6, 8])
```

```
k = np.array([False, True, True, True])
```

```
a[k]
```

# 1 次元配列 / フィルター



フィルター k を用いて、ベクトル a の要素をフィルタリングしているイメージ。

```
import numpy as np
```

```
a = np.array([ 2, 4, 6, 8])
```

```
k = np.array([True, True, False, True])
```

```
a[k]  
# array([2, 4, 8])
```

```
a = np.array([ 2, 4, 6, 8])
```

```
k = np.array([False, True, True, True])
```

```
a[k]  
# array([4, 6, 8])
```

# 1 次元配列 / フィルター

フィルターは、True または False を組み合わせて直打ちで作成できるが、ある条件を制定して、その条件に基づいて作成することが一般的である。例えば、5 よりも大きい値を取り出すフィルターや奇数の値を取り出すフィルターは、右のように作成する。

```
import numpy as np

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

f1 = (a > 5)
f1
# array([F, F, T, T, F, F, F, T])

f2 = (a % 2 == 1)
f2
# array([F, F, F, F, T, T, T, T])
```

# 1 次元配列 / フィルター

0	1	2	3	4	5	6	7	8
▼	▼	▼	▼	▼	▼	▼	▼	▼
2	4	6	8	1	3	5	7	

f1

F	F	T	T	F	F	F	T
2	4	6	8	1	3	5	7

f2

F	F	F	F	T	T	T	T
2	4	6	8	1	3	5	7

```
import numpy as np
```

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

```
f1 = (a > 5)
```

```
a[f1]
```

```
# array([6, 8, 7])
```

```
f2 = (a % 2 == 1)
```

```
a[f2]
```

```
# array([1, 3, 5, 7])
```

# 1 次元配列 / フィルター

0	1	2	3	4	5	6	7	8
▼	▼	▼	▼	▼	▼	▼	▼	▼
2	4	6	8	1	3	5	7	

f1	F	F	T	T	F	F	F	T
	2	4	6	8	1	3	5	7

f2	F	F	F	F	T	T	T	T
	2	4	6	8	1	3	5	7

a < 4	T	F	F	F	T	T	F	F
	2	4	6	8	1	3	5	7

```
import numpy as np
```

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

```
f1 = (a > 5)
```

```
a[f1]
```

```
# array([6, 8, 7])
```

```
f2 = (a % 2 == 1)
```

```
a[f2]
```

```
# array([1, 3, 5, 7])
```

```
a[(a < 4)]
```

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

フィルターを一次変数に保存せず  
に、直接使用することもできる。

# 1 次元配列 / フィルター

複数のフィルターを重ねて使用することもできる。この場合、フィルターを重ねるときに AND 演算で重ねるか、OR 演算で重ねるかを指定する必要がある。

```
import numpy as np  
  
a = np.array([1, 3, 5, 7, 9,  
             2, 4, 6, 8, 0])
```

```
f1 = (a % 2 == 1)
```

```
f2 = (a > 5)
```

```
a[f1 & f2]
```

```
a[f1 | f2]
```

# 1 次元配列 / フィルター

1	3	5	7	9	2	4	6	8	0
---	---	---	---	---	---	---	---	---	---

f1

--	--	--	--	--	--	--	--	--	--

f2

--	--	--	--	--	--	--	--	--	--

f1 & f2

--	--	--	--	--	--	--	--	--	--

1	3	5	7	9	2	4	6	8	0
---	---	---	---	---	---	---	---	---	---

f1 | f2

--	--	--	--	--	--	--	--	--	--

1	3	5	7	9	2	4	6	8	0
---	---	---	---	---	---	---	---	---	---

```
import numpy as np
```

```
a = np.array([1, 3, 5, 7, 9,  
             2, 4, 6, 8, 0])
```

```
f1 = (a % 2 == 1)
```

```
f2 = (a > 5)
```

```
a[f1 & f2]
```

```
a[f1 | f2]
```

# 1 次元配列 / フィルター

1	3	5	7	9	2	4	6	8	0
---	---	---	---	---	---	---	---	---	---

f1

T	T	T	T	T	F	F	F	F	F
---	---	---	---	---	---	---	---	---	---

f2

F	F	F	T	T	F	F	T	T	F
---	---	---	---	---	---	---	---	---	---

f1 & f2

1	3	5	7	9	2	4	6	8	0

f1 | f2

1	3	5	7	9	2	4	6	8	0

```
import numpy as np
```

```
a = np.array([1, 3, 5, 7, 9,  
             2, 4, 6, 8, 0])
```

```
f1 = (a % 2 == 1)
```

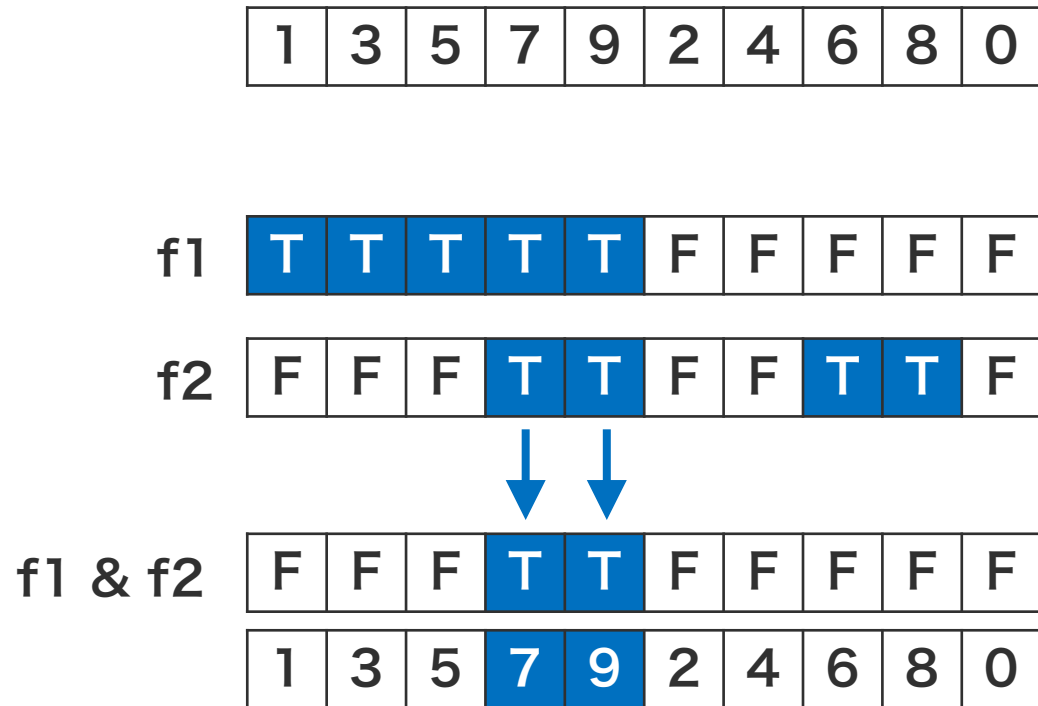
```
f2 = (a > 5)
```

```
a[f1 & f2]
```

```
a[f1 | f2]
```



# 1 次元配列 / フィルター



```
import numpy as np
```

```
a = np.array([1, 3, 5, 7, 9,  
             2, 4, 6, 8, 0])
```

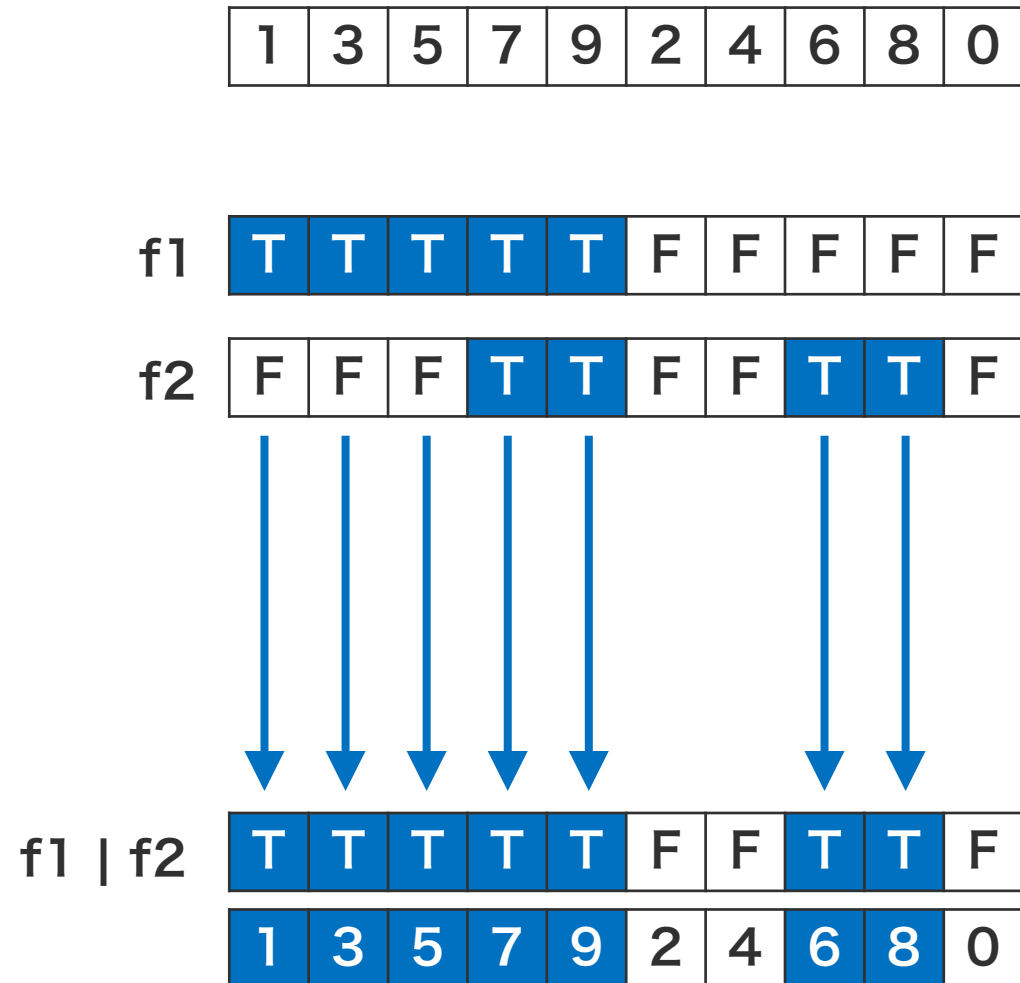
```
f1 = (a % 2 == 1)
```

```
f2 = (a > 5)
```

```
a[f1 & f2]  
# array([7, 9])
```

```
a[f1 | f2]
```

# 1 次元配列 / フィルター



```
import numpy as np
```

```
a = np.array([1, 3, 5, 7, 9,  
             2, 4, 6, 8, 0])
```

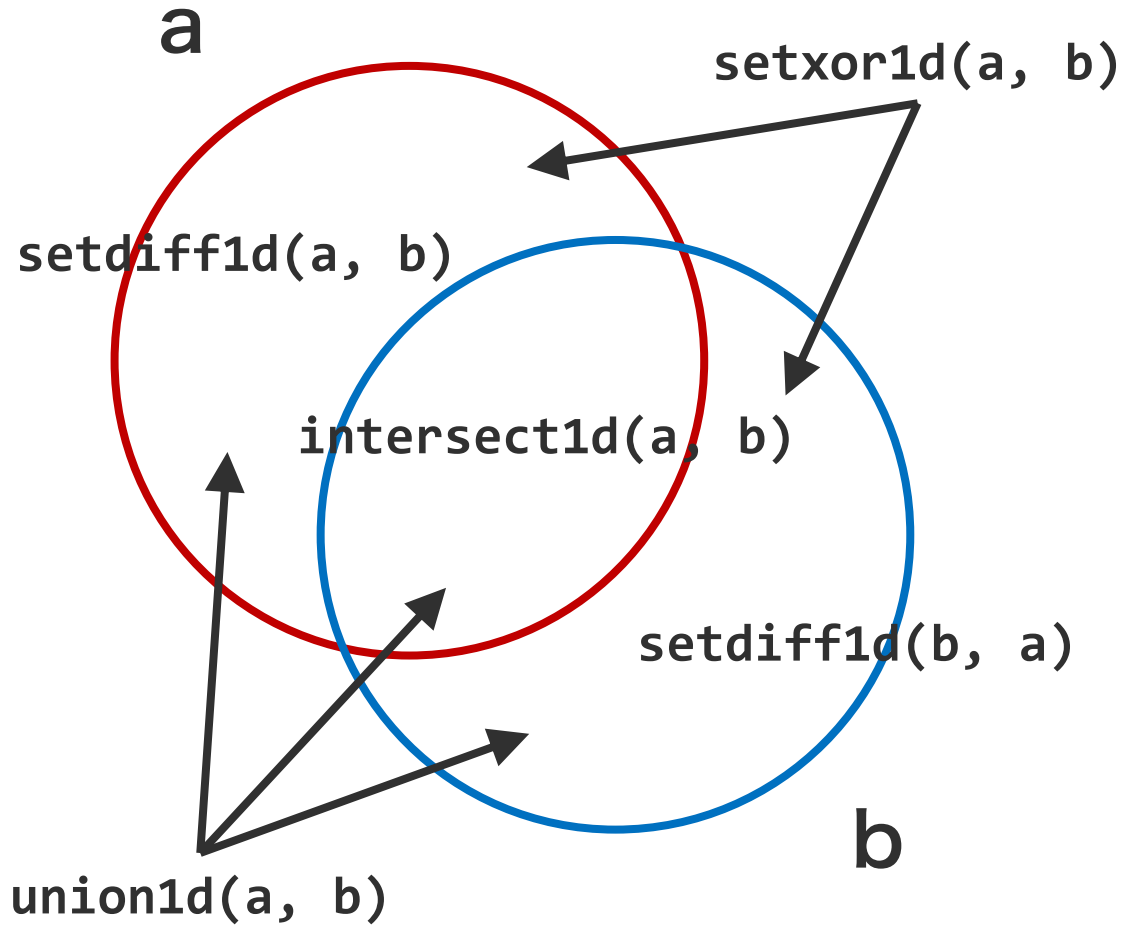
```
f1 = (a % 2 == 1)
```

```
f2 = (a > 5)
```

```
a[f1 & f2]  
# array([7, 9])
```

```
a[f1 | f2]  
# array([1, 3, 5, 7, 9, 6, 8])
```

# 集合演算



```
import numpy as np  
  
a = np.array([1, 1, 2, 3, 5, 8])  
b = np.array([2, 4, 6, 8])  
  
np.union1d(a, b)  
# array([1, 2, 3, 4, 5, 6, 8])  
  
np.setdiff1d(a, b)  
# array([1, 3, 5])  
  
np.setdiff1d(b, a)  
# array([4, 6])
```

# 擬似乱数生成

NumPy の random モジュールには擬似乱数を生成する機能が実装されている。次の表に、一様分布および正規分布から乱数を生成する関数を示した。これ以外にも、ポアソン分布やガンマ分布などの分布から乱数を生成する関数が多数用意されている。必要なときに調べて使うとよい。

メソッド	動作
<code>.rand(n)</code>	範囲 [0, 1) の一様分布から n 個の乱数を生成。
<code>.normal(m, s, n)</code>	平均 m, 標準偏差 s の正規分布から n 個の乱数を生成。
<code>.randint(l, h, n)</code>	範囲 [l, h) から n 個の整数乱数を生成。
<code>.shuffle(arr)</code>	配列 arr の要素をシャッフルする。
<code>.seed(s)</code>	乱数シードを s に固定する。

```
import numpy as np  
  
np.random.seed(2020)
```

seed を削除すると、実行するたびに異なる値が生成される。

```
np.random.rand(3)  
# array([0.986276, 0.873391, 0.509745])
```

```
np.random.normal(5, 1, 3)  
# array([4.115677, 3.762758, 4.291560])
```

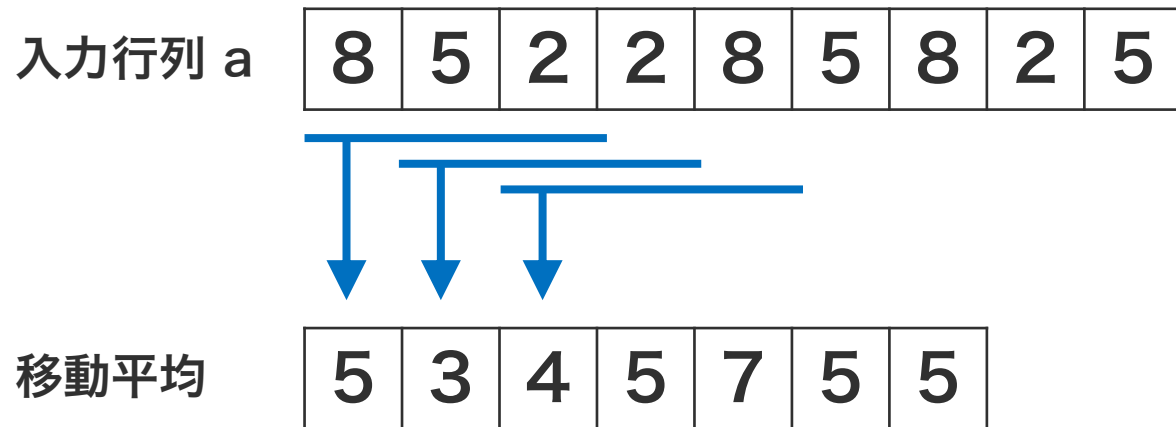
```
np.random.randint(0, 2, 5)  
# array([0, 1, 1, 1, 1])
```

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

# 問題 N1-1

🕒 10 min

入力配列  $a$  に対して 3 時点の移動平均を求めよ。



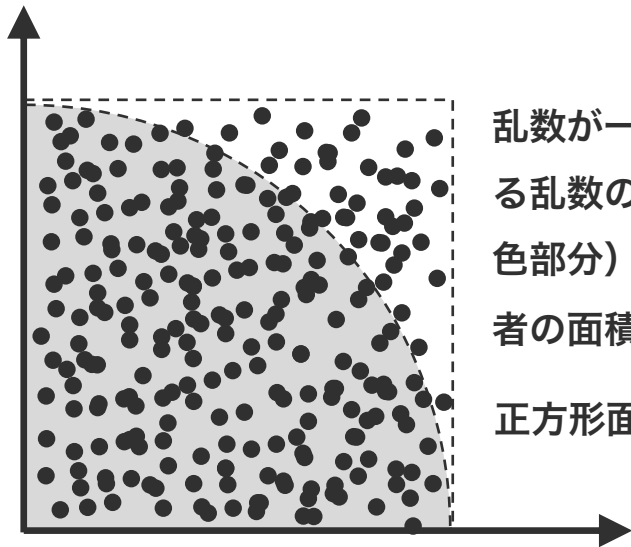
```
import numpy as np

a = np.array([8, 5, 2, 2, 8,
              5, 8, 2, 5])
```

# 問題 N1-2

🕒 15 min

`np.random.rand(n)` 関数は、 $0 \leq x < 1$  の一様分布から  $n$  個の乱数を生成する関数である。 $0 \leq x < 1$  および  $0 \leq y < 1$  の範囲で乱数を生成し、下図のように、正方形に含まれる乱数の個数と、第一象限にある単位円の内側に含まれる乱数の個数に着目して、円周率を小数 3 桁 (= 3.141...) まで正確に求めよ。



乱数が一様であれば、正方形内部に含まれる乱数の個数と第一象限にある単位円（灰色部分）に含まれる乱数の個数の比が、両者の面積の比に近似できる。

$$\text{正方形面積} : \text{灰色部分面積} = 1 : \frac{\pi}{4}$$

```
import numpy as np
```

```
n = 10000
```

```
x = np.random.rand(n)
```

```
y = np.random.rand(n)
```

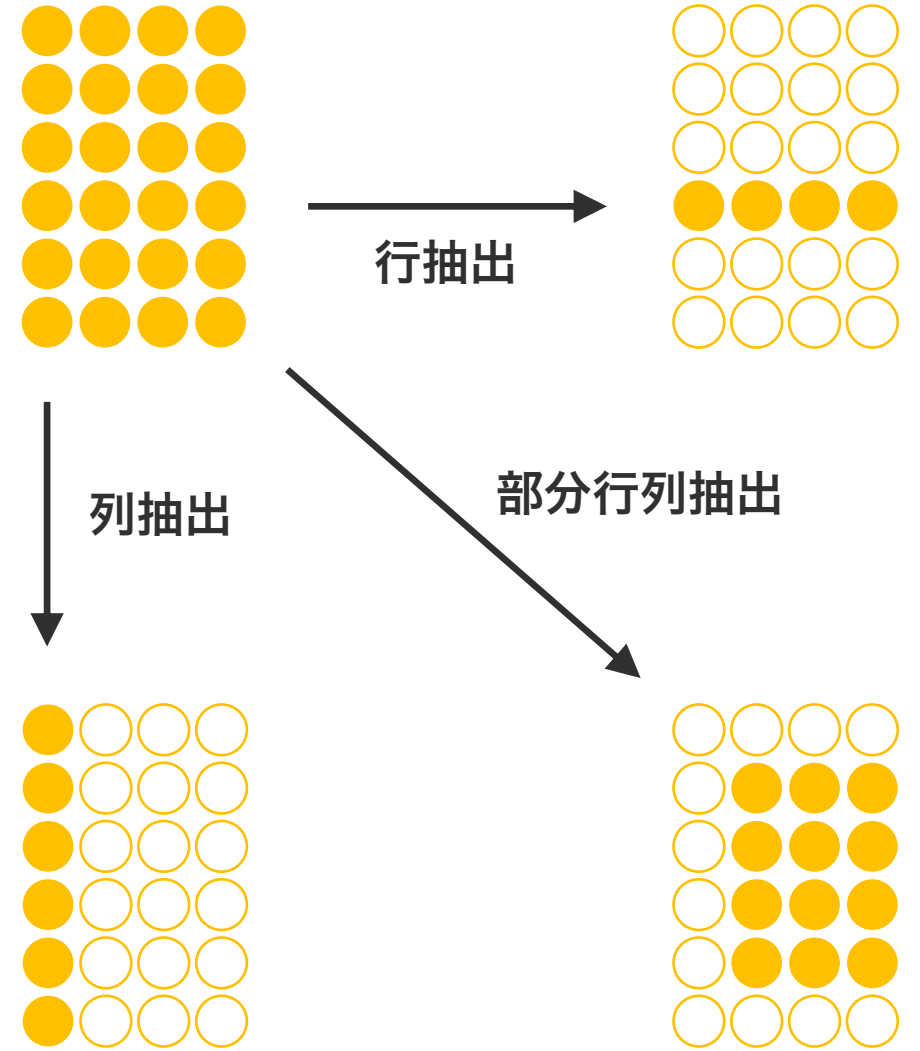
# 数値計算



- 1 次配列
- 2 次配列
- 3 次配列
- データ読み取り

# 2次元配列

NumPy の2次元配列は、縦と横の構造を持ち、数学の行列のように、特定の行あるいは列を抽出したり、行列演算を行ったりすることができる。





# 2次元配列 / np.array

2次元配列を作成するには、2次元のリストを作成して、それを np.array 関数に代入することで作成する。

```
a = np.array([[11, 12, 13, 14, 15, 16],  
              [21, 22, 23, 24, 25, 26],  
              [31, 32, 33, 34, 35, 36],  
              [41, 42, 43, 44, 45, 46],  
              [51, 52, 53, 54, 55, 56],  
              [61, 62, 63, 64, 65, 66]])
```

11	12	13	14	15	16
21	22	23	24	25	26
31	32	33	34	35	36
41	42	43	44	45	46
51	52	53	54	55	56
61	62	63	64	65	66

## 2次元配列 / np.zeros, np.ones

2次元配列も1次元配列と同様に、同じ値からなる配列を作る場合、np.zeros、np.ones、np.fullなどの関数を使う。なお、2次元配列の場合、横と縦のサイズを指定する必要がある。

```
b = np.zeros((2, 5))
```

0	0	0	0	0
0	0	0	0	0

```
b = np.ones((5, 3))
```

1	1	1
1	1	1
1	1	1
1	1	1
1	1	1

## 2次元配列 / np.full

2次元配列も1次元配列と同様に、同じ値からなる配列を作る場合、`np.zeros`、`np.ones`、`np.full`などの関数を使う。なお、2次元配列の場合、横と縦のサイズを指定する必要がある。

```
a = np.full((5, 3), np.nan)
```

nan	nan	nan
nan	nan	nan
nan	nan	nan
nan	nan	nan
nan	nan	nan

# 2次元配列 / np.identity

2次元配列の場合、np.identity 関数を使用して、単位行列を作成することができる。

```
a = np.identity(6)
```

1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0
0	0	0	0	0	1

## 2次元配列

2次元行列の構造を調べるときは、`ndim`, `shape`, `np.size` などを利用する。`ndim` 属性には、配列の次元数が記録されている。`shape` 属性には、配列構造（サイズ）が記録されている。また、`np.size` 関数を使用することで、配列の特定の次元のサイズを取得することができる。

11	12	13	14	15	16
21	22	23	24	25	26
31	32	33	34	35	36
41	42	43	44	45	46

```
a = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46]])
```

```
a.ndim
# 2
```

```
a.shape
# (4, 6)
```

```
np.size(a, axis=0)
# 4
```

```
np.size(a, axis=1)
# 6
```

## 2次元配列

2次元配列から要素を取り出すとき、位置番号を指定して取り出したり、スライスして取り出したりすることができる。

```
b = np.array([[11, 12, 13, 14, 15, 16],  
              [21, 22, 23, 24, 25, 26],  
              [31, 32, 33, 34, 35, 36],  
              [41, 42, 43, 44, 45, 46],  
              [51, 52, 53, 54, 55, 56],  
              [61, 62, 63, 64, 65, 66]])
```

```
b[1, 2]
```

```
b[1, 2:5]
```

```
b[1:5, 3]
```

```
b[2:4, 1:6]
```

## 2次元配列

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])
```

```
b[1, 2]
```

```
# 23
```

```
b[1, 2:5]
```

```
b[1:5, 3]
```

```
b[2:4, 1:6]
```

## 2 次元配列

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])
```

```
b[1, 2]
```

```
# 23
```

```
b[1, 2:5]
```

```
# array([23, 24, 25])
```

```
b[1:5, 3]
```

```
b[2:4, 1:6]
```



## 2次元配列

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],  
             [21, 22, 23, 24, 25, 26],  
             [31, 32, 33, 34, 35, 36],  
             [41, 42, 43, 44, 45, 46],  
             [51, 52, 53, 54, 55, 56],  
             [61, 62, 63, 64, 65, 66]])
```

```
b[1, 2]
```

```
# 23
```

```
b[1, 2:5]
```

```
# array([23, 24, 25])
```

```
b[1:5, 3]
```

```
# array([24, 34, 44, 54])
```

```
b[2:4, 1:6]
```

## 2次元配列

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])
```

```
b[1, 2]
# 23
```

```
b[1, 2:5]
# array([23, 24, 25])
```

```
b[1:5, 3]
# array([24, 34, 44, 54])
```

```
b[2:4, 1:6]
# array([[32, 33, 34, 35, 36],
#        [42, 43, 44, 45, 46]])
```

## 2次元配列

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])
```

```
b[3:5, 2:]
```

```
b[3:5, :]
```

```
b[:, 2]
```

```
b[:, 1:3]
```

## 2次元配列

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])
```

```
b[3:5, 2:]
# array([[43, 44, 45, 46],
#        [53, 54, 55, 56]])
b[3:5, :]
```

```
b[:, 2]
```

```
b[:, 1:3]
```

## 2次元配列

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],  
             [21, 22, 23, 24, 25, 26],  
             [31, 32, 33, 34, 35, 36],  
             [41, 42, 43, 44, 45, 46],  
             [51, 52, 53, 54, 55, 56],  
             [61, 62, 63, 64, 65, 66]])
```

```
b[3:5, 2:]  
# array([[43, 44, 45, 46],  
#        [53, 54, 55, 56]])  
b[3:5, :]  
# array([[41, 42, 43, 44, 45, 46],  
#        [51, 52, 53, 54, 55, 56]])  
b[:, 2]
```

```
b[:, 1:3]
```

## 2次元配列

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])
```

```
b[3:5, 2:]
# array([[43, 44, 45, 46],
#        [53, 54, 55, 56]])
b[3:5, :]
# array([[41, 42, 43, 44, 45, 46],
#        [51, 52, 53, 54, 55, 56]])
b[:, 2]
# array([13, 23, 33, 43, 53, 63])
```

```
b[:, 1:3]
```

## 2次元配列

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])
```

```
b[3:5, 2:]
# array([[43, 44, 45, 46],
#        [53, 54, 55, 56]])
```

```
b[3:5, :]
# array([[41, 42, 43, 44, 45, 46],
#        [51, 52, 53, 54, 55, 56]])
```

```
b[:, 2]
# array([13, 23, 33, 43, 53, 63])
```

```
b[:, 1:3]
# array([[12, 13], [22, 23], [32, 33],
#        [42, 43], [52, 53], [62, 63]])
```

## 2 次元配列 / 注意点

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],  
              [21, 22, 23, 24, 25, 26],  
              [31, 32, 33, 34, 35, 36],  
              [41, 42, 43, 44, 45, 46],  
              [51, 52, 53, 54, 55, 56],  
              [61, 62, 63, 64, 65, 66]])
```

```
r = [0, 2, 4]
```

```
c = [1, 3, 5]
```

```
b[r, c]
```



## 2 次元配列 / 注意点

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],  
             [21, 22, 23, 24, 25, 26],  
             [31, 32, 33, 34, 35, 36],  
             [41, 42, 43, 44, 45, 46],  
             [51, 52, 53, 54, 55, 56],  
             [61, 62, 63, 64, 65, 66]])
```

```
r = [0, 2, 4]
```

```
c = [1, 3, 5]
```

```
b[r, c]
```

## 2 次元配列 / 注意点

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],  
             [21, 22, 23, 24, 25, 26],  
             [31, 32, 33, 34, 35, 36],  
             [41, 42, 43, 44, 45, 46],  
             [51, 52, 53, 54, 55, 56],  
             [61, 62, 63, 64, 65, 66]])
```

```
r = [0, 2, 4]
```

```
c = [1, 3, 5]
```

```
b[r, c]
```

```
# array([12, 34, 56])
```

## 2 次元配列 / 注意点

	0	1	2	3	4	5	6
0	11	12	13	14	15	16	
1	21	22	23	24	25	26	
2	31	32	33	34	35	36	
3	41	42	43	44	45	46	
4	51	52	53	54	55	56	
5	61	62	63	64	65	66	
6							

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])
```

```
r = [0, 2, 4]
c = [1, 3, 5]
```

```
b[r, c]
# array([12, 34, 56])
```

```
b[np.ix_(r, c)]
# array([[12, 14, 16],
#        [32, 34, 36],
#        [52, 54, 56]])
```

# 行列計算

NumPy の 2 次元配列に、行列演算が定義されている。配列の各要素同士の加減乗除の他に、内積や外積も簡単に求めることができる。

計算式	計算内容
$a + b$	各要素の足し算
$a - b$	各要素の引き算
$a * b$	各要素の掛け算 (アダマール積)
$a / b$	各要素の割り算
<code>np.dot(a, b)</code>	行列同士の内積
<code>np.outer(a, b)</code>	行列同士の外積 (テンソル積)
<code>np.sum(a)</code>	全要素の和
<code>a.T</code>	行列 $a$ の転置行列

```
import numpy as np

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

b = np.array([[1, 1, 1],
              [0, 1, 1],
              [0, 0, 1]])

a + b
# array([[ 2,  3,  4],
#        [ 4,  6,  7],
#        [ 7,  8, 10]])

np.dot(a, b)
# array([[ 1,  3,  6],
#        [ 4,  9, 15],
#        [ 7, 15, 24]])
```

# 問題 N2-1

🕒 15 min

カーネル行列  $b$  を用いて、行列  $a$  に対して畳み込み演算を行い、その結果を出力せよ。

```
a = np.array([[0, 2, 0, 1, 2, 0],
              [1, 0, 3, 0, 0, 2],
              [3, 1, 0, 2, 2, 1],
              [2, 2, 2, 1, 0, 1],
              [1, 0, 1, 3, 2, 2],
              [0, 3, 1, 0, 2, 0]])
```

```
b = np.array([[1, 0, 0],
              [0, 1, 0],
              [0, 0, 1]])
```

入力行列  $a$

カーネル行列  $b$

0	2	0	1	2	0
1	0	3	0	0	2
3	1	0	2	2	1
2	2	2	1	0	1
1	0	1	3	2	2
0	3	1	0	2	0

1	0	0
0	1	0
0	0	1

$$\begin{aligned} &0*1 + 2*0 + 0*0 + \\ &1*0 + 0*1 + 3*0 + \\ &3*0 + 1*0 + 0*1 = 3 \end{aligned}$$

演算結果

0	7	2	2
4	1	5	3
6	6	3	4
3	3	7	3

# 問題 N2-1

🕒 15 min

カーネル行列  $b$  を用いて、行列  $a$  に対して畳み込み演算を行い、その結果を出力せよ。

```
a = np.array([[0, 2, 0, 1, 2, 0],  
              [1, 0, 3, 0, 0, 2],  
              [3, 1, 0, 2, 2, 1],  
              [2, 2, 2, 1, 0, 1],  
              [1, 0, 1, 3, 2, 2],  
              [0, 3, 1, 0, 2, 0]])
```

```
b = np.array([[1, 0, 0],  
              [0, 1, 0],  
              [0, 0, 1]])
```

入力行列  $a$

カーネル行列  $b$

0	2	0	1	2	0
1	0	3	0	0	2
3	1	0	2	2	1
2	2	2	1	0	1
1	0	1	3	2	2
0	3	1	0	2	0

1	0	0
0	1	0
0	0	1

$$\begin{aligned} & 2*1 + 0*0 + 1*0 + \\ & 0*0 + 3*1 + 0*0 + \\ & 1*0 + 0*0 + 2*1 = 7 \end{aligned}$$

演算結果

0	7	2	2
4	1	5	3
6	6	3	4
3	3	7	3

# 問題 N2-1

🕒 15 min

カーネル行列  $b$  を用いて、行列  $a$  に対して畳み込み演算を行い、その結果を出力せよ。

```
a = np.array([[0, 2, 0, 1, 2, 0],
               [1, 0, 3, 0, 0, 2],
               [3, 1, 0, 2, 2, 1],
               [2, 2, 2, 1, 0, 1],
               [1, 0, 1, 3, 2, 2],
               [0, 3, 1, 0, 2, 0]])
```

```
b = np.array([[1, 0, 0],
               [0, 1, 0],
               [0, 0, 1]])
```

入力行列  $a$

カーネル行列  $b$

0	2	0	1	2	0
1	0	3	0	0	2
3	1	0	2	2	1
2	2	2	1	0	1
1	0	1	3	2	2
0	3	1	0	2	0

1	0	0
0	1	0
0	0	1

$$\begin{aligned} &0*1 + 1*0 + 2*0 + \\ &3*0 + 0*1 + 0*0 + \\ &0*0 + 2*0 + 2*1 = 2 \end{aligned}$$

演算結果

0	7	2	2
4	1	5	3
6	6	3	4
3	3	7	3

# 問題 N2-2

🕒 15 min

行列  $a$  に対して  $2 \times 2$  範囲の最大プーリング演算を行い、その結果を出力せよ。

```
a = np.array([[0, 2, 0, 1, 2, 0],  
              [1, 0, 3, 0, 0, 2],  
              [3, 1, 0, 2, 2, 1],  
              [2, 2, 2, 1, 0, 1],  
              [1, 0, 1, 3, 2, 2],  
              [0, 3, 1, 0, 2, 0]])
```

入力行列  $a$

0	2	0	1	2	0
1	0	3	0	0	2
3	1	0	2	2	1
2	2	2	1	0	1
1	0	1	3	2	2
0	3	1	0	2	0

演算結果

2	3	2
3	2	2
3	3	2



# 問題 N2-2

🕒 15 min

行列  $a$  に対して  $2 \times 2$  範囲の最大プーリング演算を行い、その結果を出力せよ。

```
a = np.array([[0, 2, 0, 1, 2, 0],
              [1, 0, 3, 0, 0, 2],
              [3, 1, 0, 2, 2, 1],
              [2, 2, 2, 1, 0, 1],
              [1, 0, 1, 3, 2, 2],
              [0, 3, 1, 0, 2, 0]])
```

入力行列  $a$

0	2	0	1	2	0
1	0	3	0	0	2
3	1	0	2	2	1
2	2	2	1	0	1
1	0	1	3	2	2
0	3	1	0	2	0

演算結果

2	3	2
3	2	2
3	3	2

# 問題 N2-2

🕒 15 min

行列 a に対して 2x2 範囲の最大プーリング演算を行い、その結果を出力せよ。

```
a = np.array([[0, 2, 0, 1, 2, 0],
              [1, 0, 3, 0, 0, 2],
              [3, 1, 0, 2, 2, 1],
              [2, 2, 2, 1, 0, 1],
              [1, 0, 1, 3, 2, 2],
              [0, 3, 1, 0, 2, 0]])
```

入力行列 a

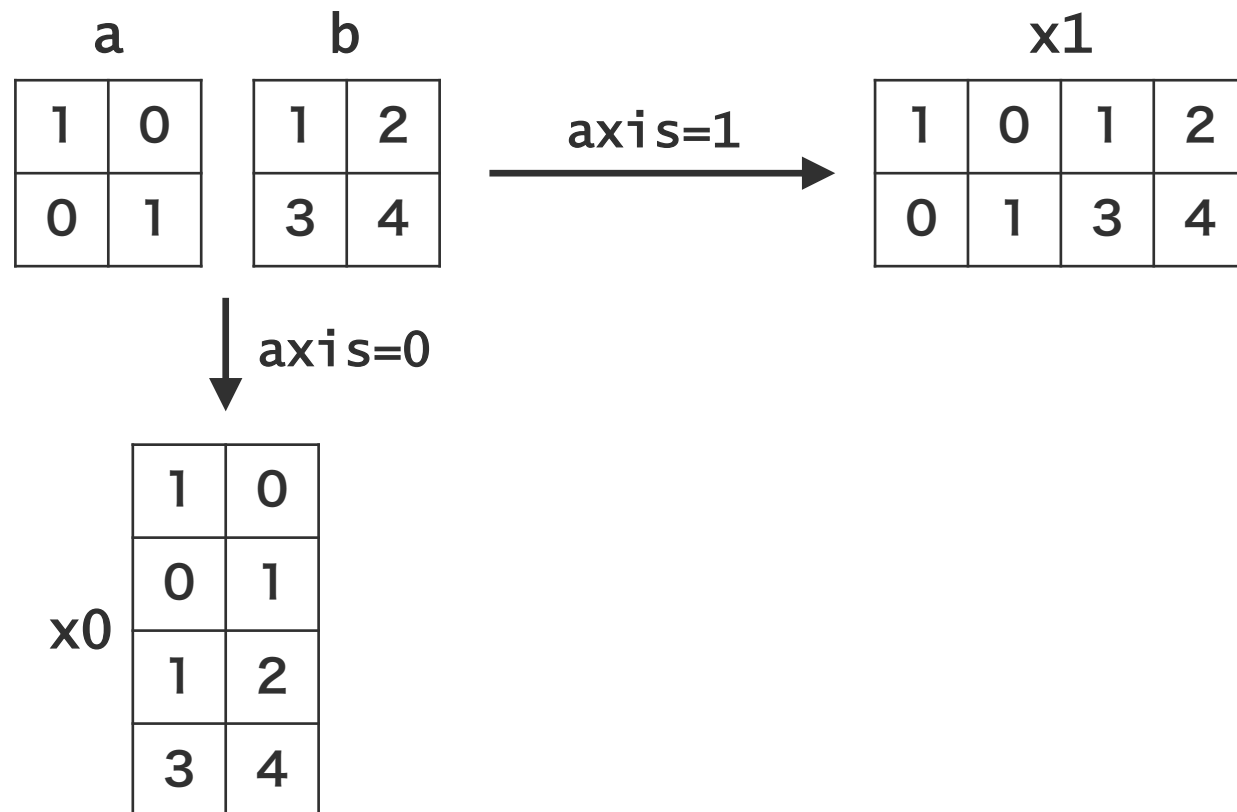
0	2	0	1	2	0
1	0	3	0	0	2
3	1	0	2	2	1
2	2	2	1	0	1
1	0	1	3	2	2
0	3	1	0	2	0

演算結果

2	3	2
3	2	2
3	3	2

# 行列操作 / np.concatenate

np.concatenate 関数は、複数のNumPy 配列を結合する機能を持つ。この関数を使うとき、結合する軸（次元）方向を axis 引数で指定。省略された場合は axis=0 となる。



```
a = [[1, 0], [0, 1]]
b = [[1, 2], [3, 4]]
```

```
x0 = np.concatenate([a, b], axis=0)
x0
```

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

```
x1 = np.concatenate([a, b], axis=1)
x1
```

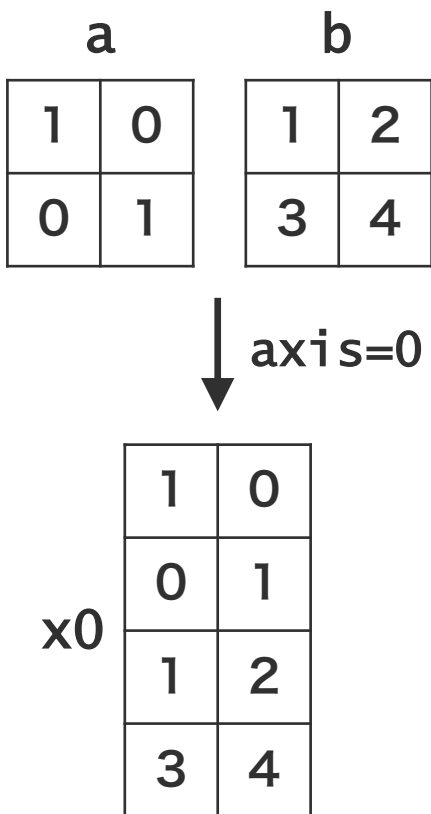
```
# array([[1, 0, 1, 2],
#        [0, 1, 3, 4]])
```

```
c = [9, 9]
```

```
y0 = np.concatenate([a, c], axis=0)
# ValueError: all the input arrays must
# have same number of dimensions, ...
```

# 行列操作 / np.vstack

np.vstack 関数は複数の配列を第 1 次元方向に結合する機能を持つ。axis=0 のときの np.concatenate 関数の機能と同じ。



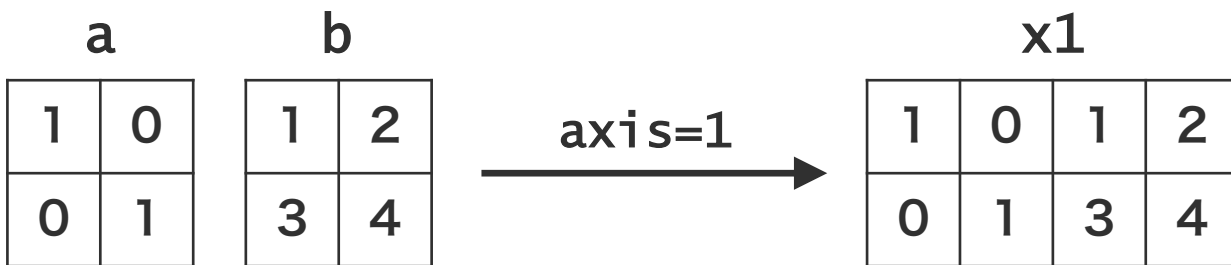
```
a = [[1, 0], [0, 1]]  
b = [[1, 2], [3, 4]]
```

```
x = np.concatenate([a, b], axis=0)  
x  
# array([[1, 0],  
#        [0, 1],  
#        [1, 2],  
#        [3, 4]])
```

```
y = np.vstack([a, b])  
y  
# array([[1, 0],  
#        [0, 1],  
#        [1, 2],  
#        [3, 4]])
```

# 行列操作 / np.hstack

np.hstack 関数は複数の配列を第 2 次元方向に結合する機能を持つ。axis=1 のときの np.concatenate 関数の機能と同じ。



```
a = [[1, 0], [0, 1]]  
b = [[1, 2], [3, 4]]
```

```
x = np.concatenate([a, b], axis=1)  
x  
# array([[1, 0, 1, 2],  
#        [0, 1, 3, 4]])
```

```
y = np.hstack([a, b])  
y  
# array([[1, 0, 1, 2],  
#        [0, 1, 3, 4]])
```

# 行列操作 / flatten

flatten 関数は、多次元配列を 1 次元配列に変更する関数である。多次元配列を崩すときに、横方向から崩すのか (order='C')、縦方向から崩すのか (order='F') を指定することもできる。

```
a = np.array([[1, 0, 2, 4],
              [0, 1, 3, 9]])

x = a.flatten()
x
# array([1, 0, 2, 4, 0, 1, 3, 9])

y = a.flatten(order='F')
y
# array([1, 0, 0, 1, 2, 3, 4, 9])
```

# 行列操作 / ravel

ravel 関数は、flatten 関数とほぼ同じように使う。ただ、ravel 関数の場合、変更した 1 次元配列は、元の多次元配列のビューであるため（見た目は異なるが、メモリを共有）、どちらか一方の値を変更すると、他方も変更されてしまう。

```
a = np.array([[1, 0, 2, 4],  
              [0, 1, 3, 9]])
```

```
x = a.ravel()
```

```
x
```

```
# array([1, 0, 2, 4, 0, 1, 3, 9])
```

```
x[2] = -1
```

```
a
```

```
# array([[ 1,  0, -1,  4],  
#        [ 0,  1,  3,  9]])
```

```
a[0, 0] = -1
```

```
x
```

```
# array([-1,  0, -1,  4,  0,  1,  3,  9])
```

# 行列操作 / reshape

reshape 関数は、多次元配列の構造を変形する関数である。reshape の 1 つ目の引数に変形後の構造をリストで指定する。また、order 引数を指定することで、多次元配列のデータを読み取る方向を指定することもできる。

reshape 関数を利用して変形した配列は、元の配列のビューまたはコピーとなっている。メモリー容量や配列のサイズによってビューかコピーが決定される。したがって、変形後の配列の値を変更すると、変形元の配列の値も変わる可能性がある。

```
a = np.array([[1, 2, 2, 3],
              [0, 1, 2, 6],
              [0, 0, 1, 9]])

y = a.reshape([12])
y
## array([1, 2, 2, 3, 0, 1, 2,
##        6, 0, 0, 1, 9])

x = a.reshape([4, 3])
x
## array([[1, 2, 2],
##        [3, 0, 1],
##        [2, 6, 0],
##        [0, 1, 9]])
```



# 行列操作 / reshape

reshape 関数は、多次元配列の構造を変形する関数である。reshape の 1 つ目の引数に変形後の構造をリストで指定する。また、order 引数を指定することで、多次元配列のデータを読み取る方向を指定することもできる。

reshape 関数を利用して変形した配列は、元の配列のビューまたはコピーとなっている。メモリー容量や配列のサイズによってビューかコピーが決定される。したがって、変形後の配列の値を変更すると、変形元の配列の値も変わる可能性がある。

```
a = np.array([[1, 2, 2, 3],
              [0, 1, 2, 6],
              [0, 0, 1, 9]])
```

```
y = a.reshape([12], order='F')
y
## array([1, 0, 0, 2, 1, 0, 2,
##        2, 1, 3, 6, 9])
```

```
x = a.reshape([4, 3], order='F')
x
## array([[1, 1, 1],
##        [0, 0, 3],
##        [0, 2, 6],
##        [2, 2, 9]])
```

# 行列操作 / reshape

reshape 関数にて、変形後の配列の構造を指定するときに、-1 を指定することができる。-1 で指定された次元のサイズは、他の次元のサイズから自動的に計算される。

```
a = np.array([[1, 2, 2, 3],
              [0, 1, 2, 6],
              [0, 0, 1, 9]])
```

```
y1 = a.reshape([2, 2, 3])
```

```
y1
# array([[[1, 2, 2],
#         [3, 0, 1]],
#        [[2, 6, 0],
#         [0, 1, 9]]])
```

```
y2 = a.reshape([2, 2, -1])
```

```
y2
# array([[[1, 2, 2],
#         [3, 0, 1]],
#        [[2, 6, 0],
#         [0, 1, 9]]])
```

a は 12 要素。最初の 2 次元に 2 要素ずつを配置した場合、最後の次元は自動的に 3 になる。

# 行列操作 / np.squeeze

np.squeeze 関数は、サイズが 1 の次元を削除する機能を持つ。np.squeeze は、元の配列のビューを返すので、変形後の要素を編集するときに十分に注意すること。

```
a = np.arange(20).reshape(2, 2, 1, 5)
```

```
a  
# array([[[[ 0,  1,  2,  3,  4]],  
#        [[ 5,  6,  7,  8,  9]]],  
#        [[[10, 11, 12, 13, 14]],  
#        [[15, 16, 17, 18, 19]]])
```

```
a.shape  
# (2, 2, 1, 5)
```

```
x = a.squeeze()  
x.shape  
# (2, 2, 5)
```

# 問題 N2-3

🕒 20 min

下図のように入力配列 a に対して、畳み込み演算と最大プーリング演算を行い、最後にその結果を 1 次元配列に変更せよ。なお、演算方法については問題 N2-1, N2-2 を参照のこと。

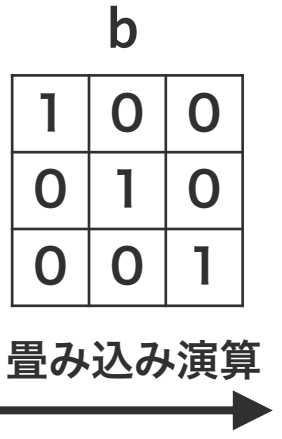
```
a = np.array([[0, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 2, 0, 1, 2, 0, 0],
              [0, 1, 0, 3, 0, 0, 2, 0],
              [0, 3, 1, 0, 2, 2, 1, 0],
              [0, 2, 2, 2, 1, 0, 1, 0],
              [0, 1, 0, 1, 3, 2, 2, 0],
              [0, 0, 3, 1, 0, 2, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0]])
```

```
b = np.array([[1, 0, 0],
              [0, 1, 0],
              [0, 0, 1]])
```

X
5
6
4
7
6
7
4
4
5

a

0	0	0	0	0	0	0	0
0	0	2	0	1	2	0	0
0	1	0	3	0	0	2	0
0	3	1	0	2	2	1	0
0	2	2	2	1	0	1	0
0	1	0	1	3	2	2	0
0	0	3	1	0	2	0	0
0	0	0	0	0	0	0	0



c

0	5	0	1	4	0
2	0	7	2	2	4
5	4	1	5	3	1
2	6	6	3	4	3
4	3	3	7	3	2
0	4	1	1	5	2



d

5	7	4
6	6	4
4	7	5



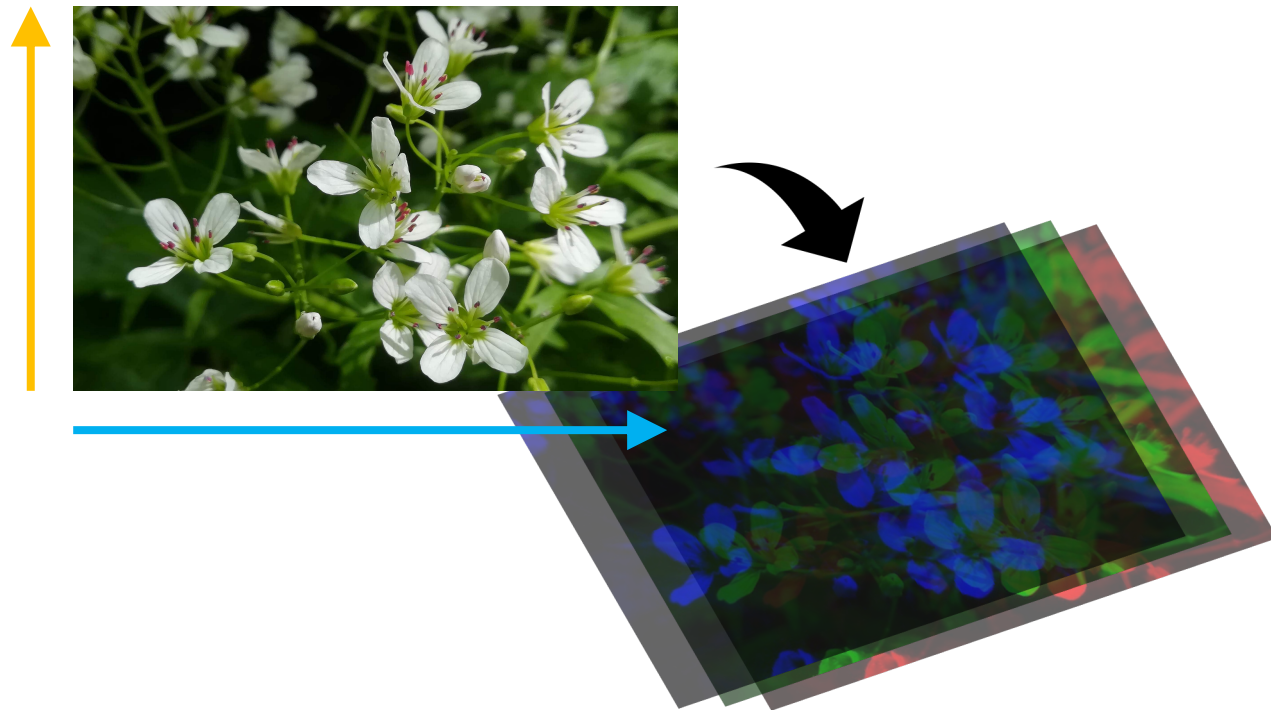
# 数値計算



- 1 次配列
- 2 次配列
- 3 次配列
- データ読み取り

# 3次元配列

3次元配列は、画像解析のときによく使われる。デジタルカメラで撮られている画像の色は、赤・緑・青の3原色によって表される。そのため、縦  $n$  横  $m$  の画像は、 $n \times m$  の配列がまずあり、その各  $(n, m)$  要素にはRGBの3つの要素が含まれているようになる。



```
b = np.array([[1, 0, 1], [1, 1, 0], [1, 1, 1]],
              [[1, 0, 1], [1, 1, 0], [1, 1, 1]],
              [[1, 0, 1], [1, 1, 0], [1, 1, 1]],
              [[1, 0, 1], [1, 1, 0], [1, 1, 1]],
              [[1, 0, 1], [1, 1, 0], [1, 1, 1]])
b
array([[1 0 1]
       [1 1 0]
       [1 1 1]]
       [[1 0 1]
       [1 1 0]
       [1 1 1]]
       [[1 0 1]
       [1 1 0]
       [1 1 1]]
       [[1 0 1]
       [1 1 0]
       [1 1 1]]
       [[1 0 1]
       [1 1 0]
       [1 1 1]])
```

# 多次元配列

---

多次元配列は、深層学習による画像分類のときに使われることが多い。深層学習では、一度に画像を 16~64 枚ずつまとめて学習するので、3 次元の画像を複数枚束ねたときに、4 次元の配列が使われる。

# 数値計算



- 1 次配列
- 2 次配列
- 3 次配列
- データ読み取り



# データの書き出し / バイナリ

NumPy の配列データをテキストファイルに書き出して保存する方法は、2 通りある。1 つは、配列データをバイナリ形式で書き出す方法である。この場合、データの構造やメタ情報も同時に書き出されるため、情報の欠損が少ない。1 つの配列を 1 つのファイルに保存するとき、save 関数を使う。複数個の配列を 1 つのファイルに保存するとき、savez を使う。

```
<93>NUMPY^A^v^@{'descr':  
'<i8', 'fortran_order':  
False, 'shape': (5,)}, }  
^A^@^A^@^A^@^A^@^A^B^@^A^@^A^@  
^A^@^A^C^@^A^@^A^@^A^@^A^D^@^A^@  
^A^@^A^@^A^@^E^@^A^@^A^@^A^@
```

.npy

```
# binary (.npy)
```

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

```
np.save('data.npy', a)
```

```
# binary (.npz)
```

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

```
b = np.array([0, 0, 1, 0, 1])
```

```
np.savez('objects.npz', a = a, b = b)
```

# データの読み込み / バイナリー

NumPy のバイナリデータを保存しているファイルからデータを読み込むときに `load` 関数を使用する。複数の配列を含むファイルを読み込むときも `load` 関数を使用するが、その際、複数個の配列はディクショナリに似たデータ構造でオブジェクトに代入される。そのオブジェクトの `.files` 属性には、配列の名前の一覧が保存されている。

```
# binary (.npy)
a = np.load('data.npy')
a
# array([1, 2, 3, 4, 5])

# binary (.npz)
x = np.load('objects.npz')
x.files
# ['a', 'b']

x['a']
# array([1, 2, 3, 4, 5])

x['b']
# array([0, 0, 1, 0, 1])
```

# データの書き出し / テキスト

NumPy の配列データをファイルに保存するもう 1 つの方法は、配列データをテキストデータとして書き出す方法である。可読性はあるものの、桁数の多い小数などを正確に保存できないことがある。テキストファイルとして書き出すとき、`savetxt` 関数を使用する。この関数のオプションを指定することで、データ間の区切り文字や小数の有効桁数を指定することができる。

```
1.0000000000000000e+00
2.0000000000000000e+00
3.0000000000000000e+00
4.0000000000000000e+00
5.0000000000000000e+00
```

`.txt`

```
# text (.txt)

a = np.array([1, 2, 3, 4, 5])

np.savetxt('data.tsv', a)

np.savetxt('data.csv', a,
           fmt='%.18e',
           delimiter=',')
```

# データの読み込み / テキスト

テキストファイルを読み込むときに `loadtxt` 関数または `genfromtxt` 関数を使用する。`loadtxt` 関数を使用するとき、読み込み時にエラーが発生した場合、ファイルの区切り文字やヘッダーに合わせて、`loadtxt` 関数のオプションを調整することで対処できる。また、`loadtxt` 関数は欠損値を含むファイルを処理できないため、欠損値を含む場合、`genfromtxt` 関数を使用する。

```
# text (.txt)

a = np.loadtxt('data.txt')
a
# array([1., 2., 3., 4., 5.])
```

# 問題 N3-1

---

iris.txt はタブ区切りのテキストファイルであり、3 種のアヤメ (setosa, versicolor, virginica) の萼 (sepal) と花弁 (petal) の長さ と幅のデータが記載されている。NumPy の機能を使って、ファイルを読み込み、setosa, versicolor, および virginica それぞれの花弁の長さの平均を求めよ。なお、iris.txt の一行目はヘッダ行であることに注意せよ。

```
f = 'iris.txt'
```

 <https://aabbdd.jp/data/iris.txt>

---