農学生命情報科学特論 I

ICT や IoT 等の先端技術を活用し、効率よく高品質生産を可能にするスマート農業への取り組みは世界的に進められています。その基礎を支えている技術の一つがプログラミング言語。なかでも、習得しやすくかつ応用範囲の広い Python がとくに注目されています。本科目では、農学や分子生物学などの分野で利用される Python の最新事例を紹介しながら、Python の基礎文法の講義を行います。

孫　建強　https://aabbdd.jp/
農研機構・農業情報研究センター

June
06 17:15–20:30

基本構文

第 1 回目の授業では、プログラミング言語の基本であるデータ構造とアルゴリズムを簡単に紹介してから、Python の基本構文を紹介する。Python のスカラー、リスト、ディクショナリ、条件構文と繰り返し構文を取り上げる。

June
13 17:15–20:30

文字列処理

バイオインフォマティックスの分野において、塩基配列やアミノ酸配列などの文字列からなるデータを扱うことが多い。第 2 回目の授業では、Python を利用した文字列処理を紹介し、FASTA や GFF などのファイルから情報を抽出する方法を取り上げる。

June
20 17:15–20:30

データ解析

Python で CSV や TSV ファイルに保存された数値データを扱うときに、NumPy や Pandas ライブラリーを使用する。第 3 回目の授業では、NumPy や Pandas の機能を紹介し、ベクトルや行列のデータ処理を中心に取り上げる。

June
27 17:15–20:30

データ可視化

Python で数値データを可視化するときに matplotlib ライブラリーを使用する。第 4 回目の授業では、matplotlib の基本的な使い方を紹介し、全回の授業を復習しながらデータの可視化を行う。

レポートの提出方法

- レポートを Jupyter Notebook / Jupyter Lab 上で解き、そのファイル（.ipynb）を メールで提出する。

  ✉ report＠aabbdd.jp

- 注意事項
  - メールの件名を「特論I課題3」としてください。
  - メールの本文に何も書かないでください。
  - ファイル名を "英語氏名-3.ipynb"（例： ShinosukeNohara-3.ipynb）としてください。
  - ファイル中に、名前・所属・研究内容などの個人情報・機密情報を書かないでください。個人情報・機密情報を含むレポートを零点とする。
  - 締切日：2022-06-30

レポート問題（第 3 回）

問題 1

サイズが同じである 2 つの 2 次元配列 x および y の要素を比較し、大きい方の要素を選んで新しい配列 z を作成せよ。NumPy の利用を想定しているが、使用しなくてもよい。 （5 点）

x

6	7	2
3	0	4
9	5	1

y

0	3	5
7	1	2
4	8	6

z

6	7	5
7	1	4
9	8	6

レポート問題（第 3 回）

問題 2

sleep_in_mammals.txt ファイルを読み込み、TotalSleep が 8 時間以上の生物は何種あるかを調べるプログラムを作成せよ。Pandas の利用を想定しているが、使用しなくてもよい。（5 点）

レポート問題（第 3 回）

問題 3

tair データベースからシロイヌナズナ (*A. thaliana*) の遺伝子アノテーション TAIR10_GFF3_genes.gff をダウンロードしてください。補足情報を参考にして、アノテーションに記載されたすべての mRNA の長さを取得し、その平均値を計算せよ。 （5点）

- TAIR10_GFF3_genes.gff (44,139KB; 2019-07-11) ダウンロード先：arabidopsis.org
- GFF ファイルはタブ区切りのファイルで、Pandas で読み込むのが簡単。ただし、Pandas を使わなくても良い。
- プログラムは、平均値のみを出力するようにしてください。

問題 3 補足

```
>>> x = pd.read_csv('TAIR10_GFF3_genes.gff', sep='\t', header=None)
>>> x.head(20)
       0       1                2     3         4  5  6  7                                                  8
0   Chr1  TAIR10        chromosome     1  30427671  .  .  .                                      ID=Chr1;Name=Chr1
1   Chr1  TAIR10             gene  3631      5899  .  +  .  ID=AT1G01010;Note=protein_coding_gene;Name=AT1...
2   Chr1  TAIR10             mRNA  3631      5899  .  +  .  ID=AT1G01010.1;Parent=AT1G01010;Name=AT1G01010...
3   Chr1  TAIR10          protein  3760      5630  .  +  .  ID=AT1G01010.1-Protein;Name=AT1G01010.1;Derive...
4   Chr1  TAIR10             exon  3631      3913  .  +  .                                   Parent=AT1G01010.1
5   Chr1  TAIR10    five_prime_UTR  3631      3759  .  +  .                                   Parent=AT1G01010.1
6   Chr1  TAIR10              CDS  3760      3913  .  +  0         Parent=AT1G01010.1,AT1G01010.1-Protein;
:      :      :                :     :         :  :  :  :                                                  :
15  Chr1  TAIR10             exon  5439      5899  .  +  .                                   Parent=AT1G01010.1
16  Chr1  TAIR10              CDS  5439      5630  .  +  0         Parent=AT1G01010.1,AT1G01010.1-Protein;
17  Chr1  TAIR10  three_prime_UTR  5631      5899  .  +  .                                   Parent=AT1G01010.1
18  Chr1  TAIR10             gene  5928      8737  .  -  .  ID=AT1G01020;Note=protein_coding_gene;Name=AT1...
19  Chr1  TAIR10             mRNA  5928      8737  .  -  .  ID=AT1G01020.1;Parent=AT1G01020;Name=AT1G01020...
```

開始位置　　終了位置

3列目がmRNAならば、この行はmRNAのアノテーションであることを示している。

この差が長さである

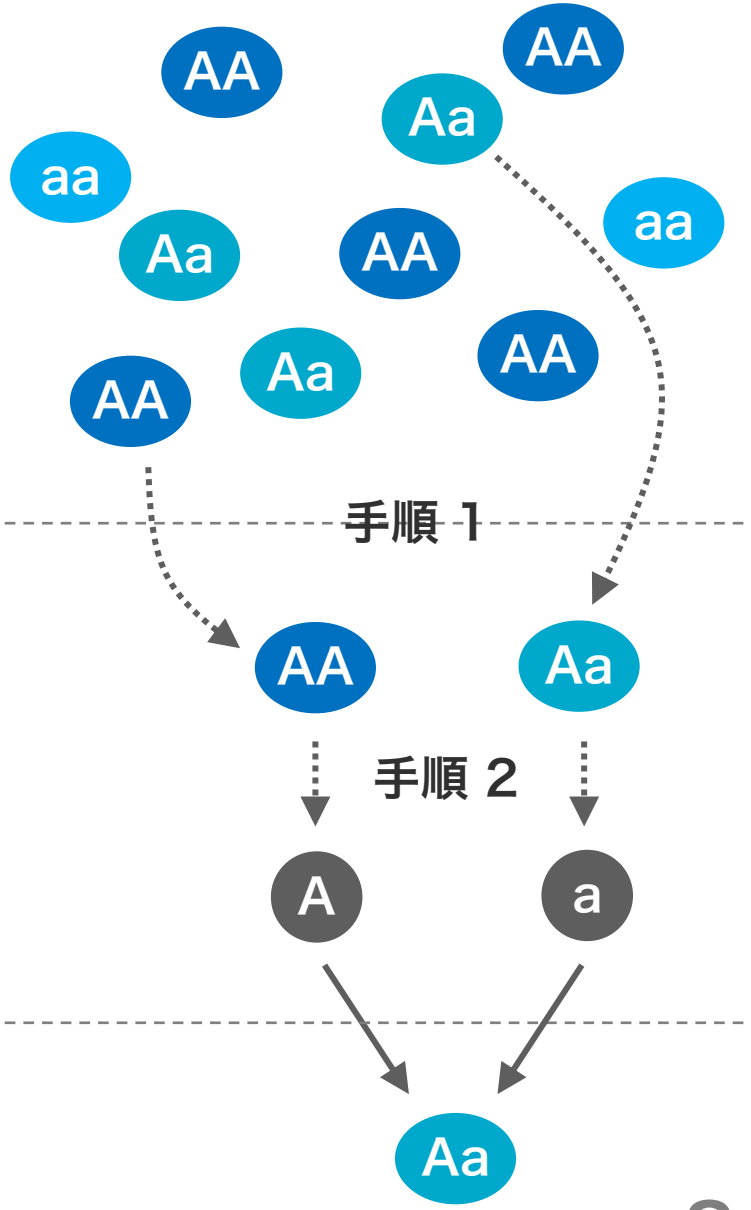
レポート問題（第 3 回）

問題 4

ある二倍体生物からなる集団の中で、遺伝子型 AA、Aa、aa を持つ個体がそれぞれ 600 個体、300 個体、100 個体存在する。この集団において、次のような操作を行う。 （10 点）

1. この集団の中からランダムに 2 個体を抽出する（非復元抽出）。
2. 抽出した 2 個体それぞれからランダムにアレル（A または a）を 1 つ抽出し、両者を合わせて次世代の個体のアレルとする。
3. 手順 1-2 を 1000 回繰り返して、計 1000 個体を生成する。

手順 1-3 を NumPy や乱数機能を使って実装し、手順 3 で得られた 1000 個体中の遺伝子型 AA、Aa、aa 個体の数をそれぞれ求めよ。



農学生命情報科学特論Ⅰ

3

- NumPy
- Pandas

NumPy

- １ 次元配列
- ２ 次元配列
- ３ 次元配列
- データの読み書き

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 arrayrange 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。

```
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.  ])
```

np.array

np.zeros　np.ones　np.full

np.arange

np.linspace

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

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
```

https://docs.scipy.org/doc/numpy/reference/ufuncs.html

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])
```

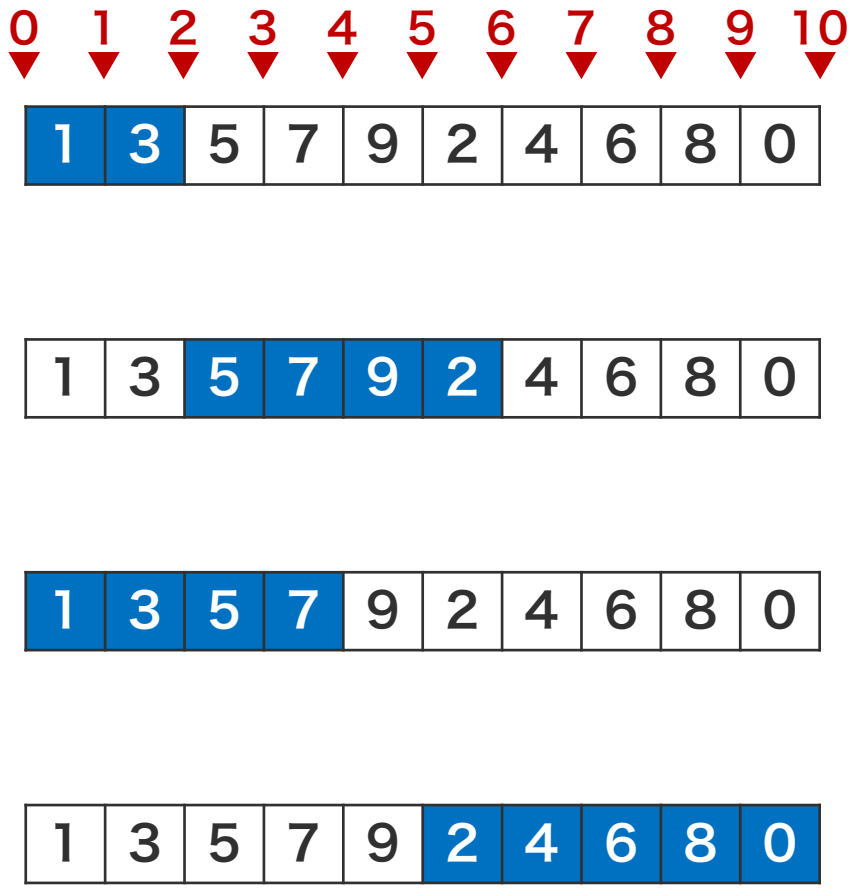
1 次元配列 / スライス

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。

```
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])
```

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。

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])
```

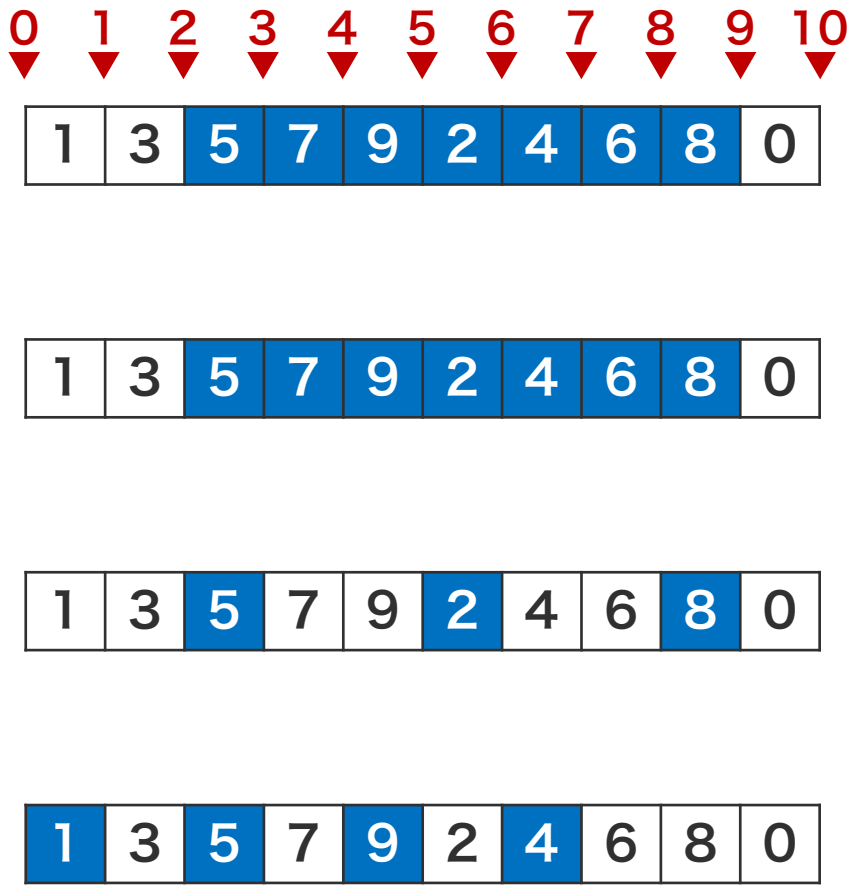
```
a[5:]
# np.array([2, 4, 6, 8, 0])
```

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])
```

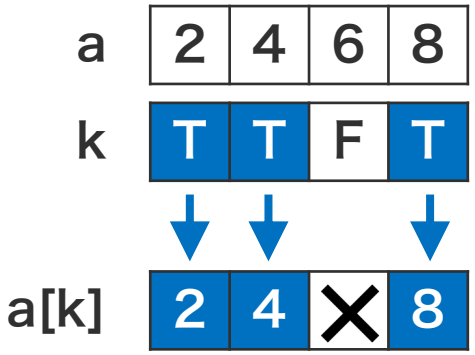
1 次元配列 / フィルター

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]
```

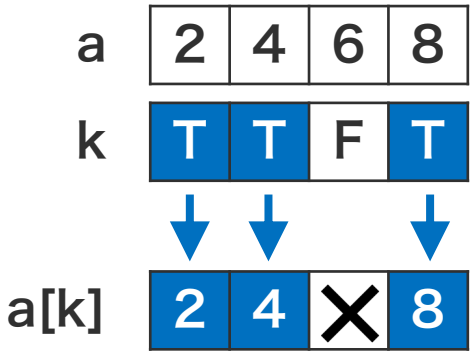
1 次元配列 / フィルター

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。

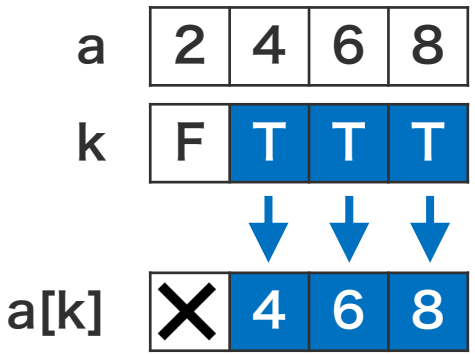



```
import numpy as np


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




```
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])
```

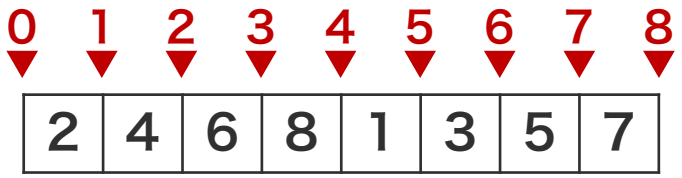
フィルター k を用いて、ベクトル a の要素をフィルタリングしているイメージ。

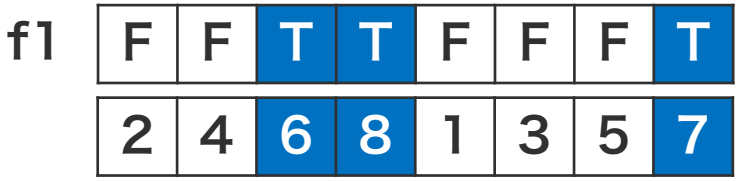
1 次元配列 / フィルター

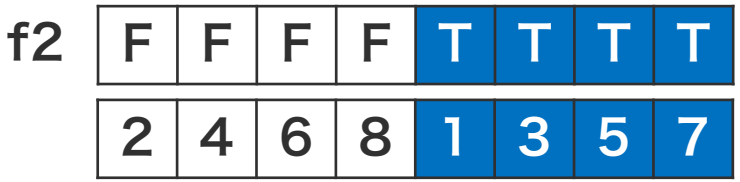
フィルターは、True または False を組み合わせて直打ちで作成できるが、ある条件を制定して、その条件に基づいて作成することが一般的である。例えば、5 よりも大きい値を取り出すフィルターや奇数の値を取り出すフィルターは、右のように作成する。

```
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

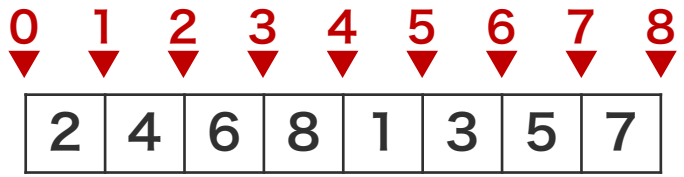


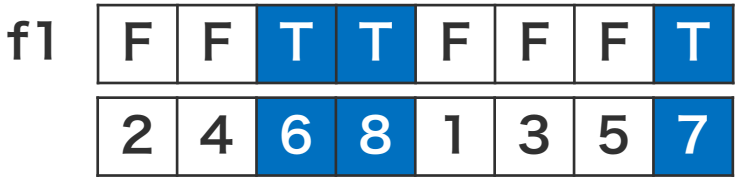


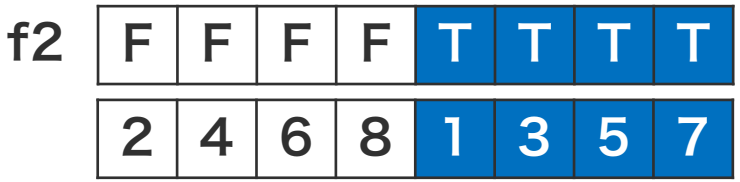


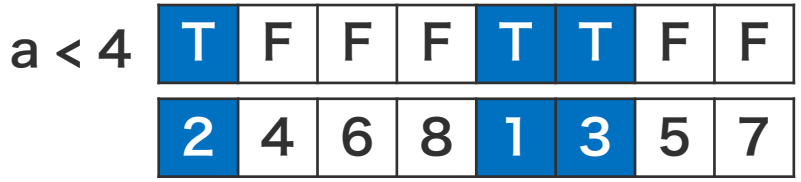

```
import numpy as np


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










```
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])
```

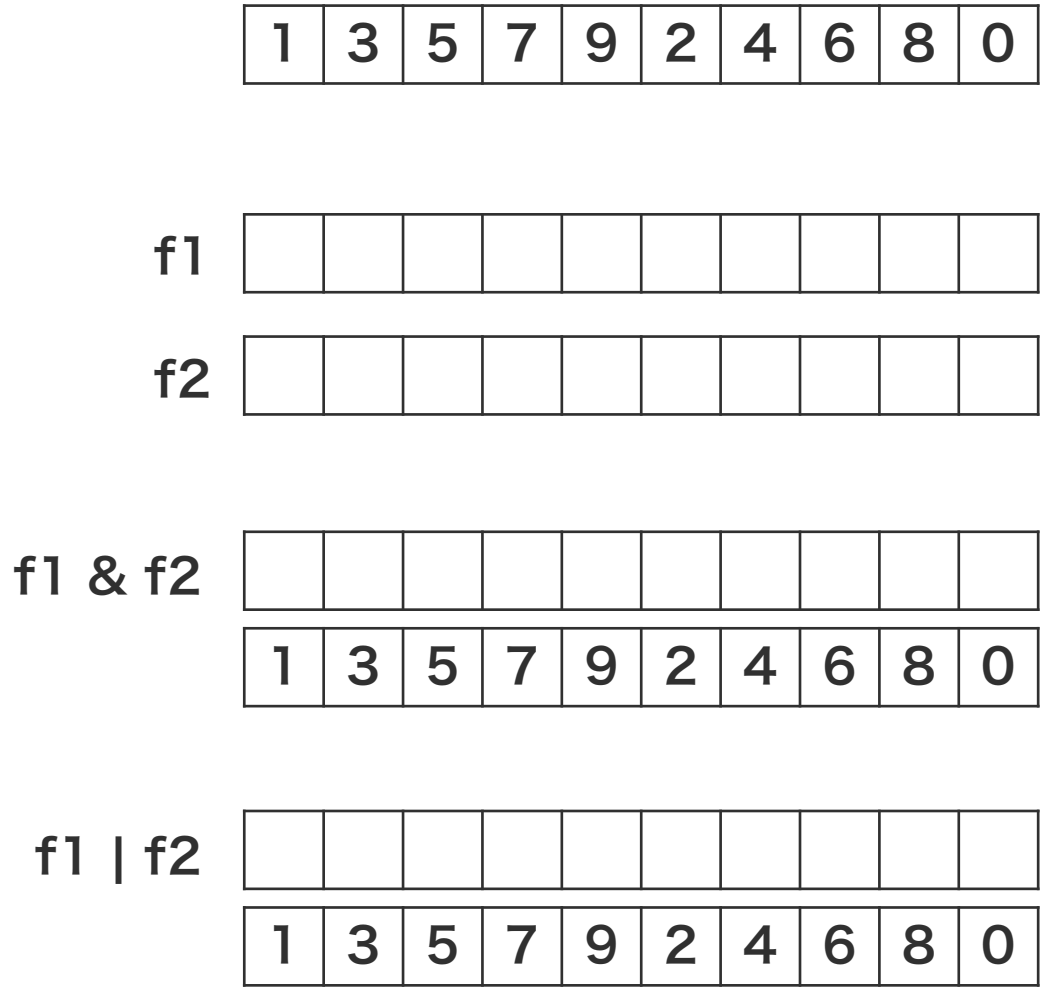
フィルターを一次変数に保存せずに、直接使用することもできる。

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。

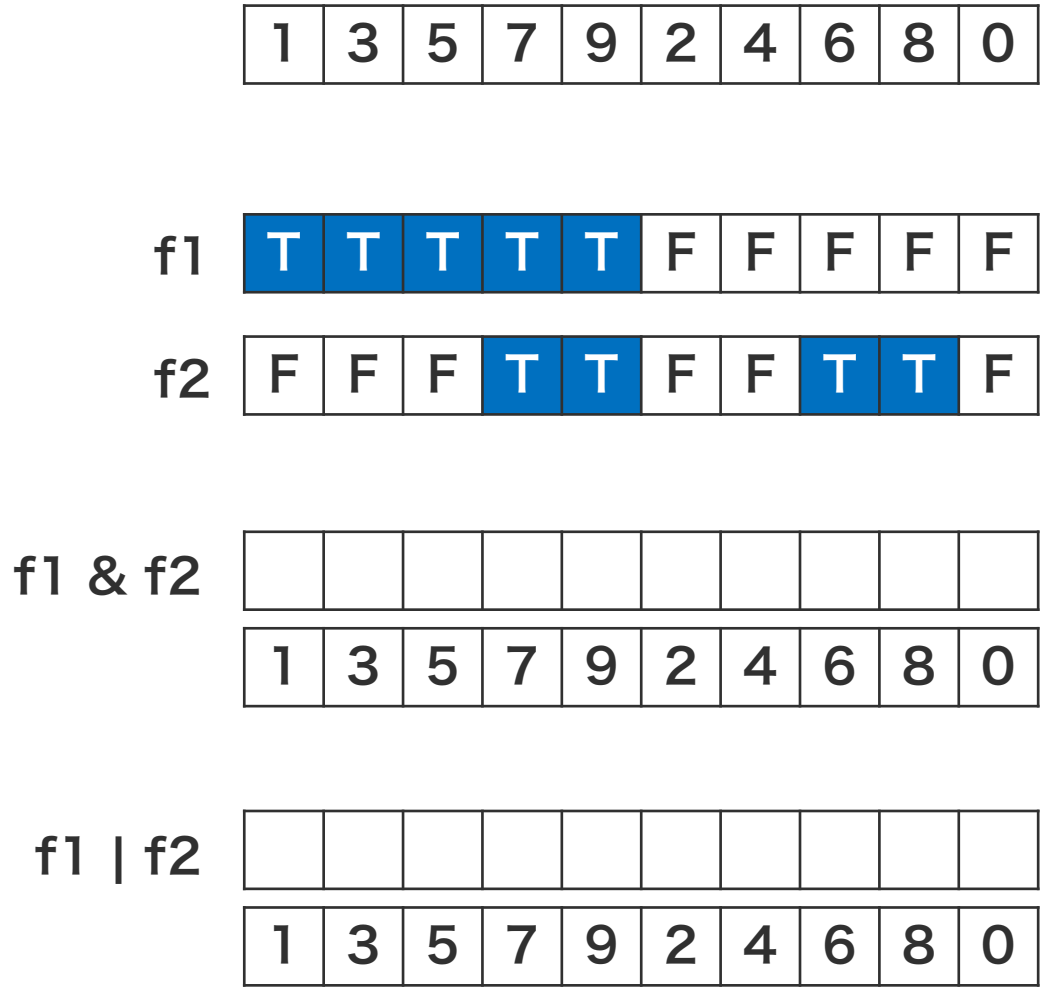
```
import numpy as np

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



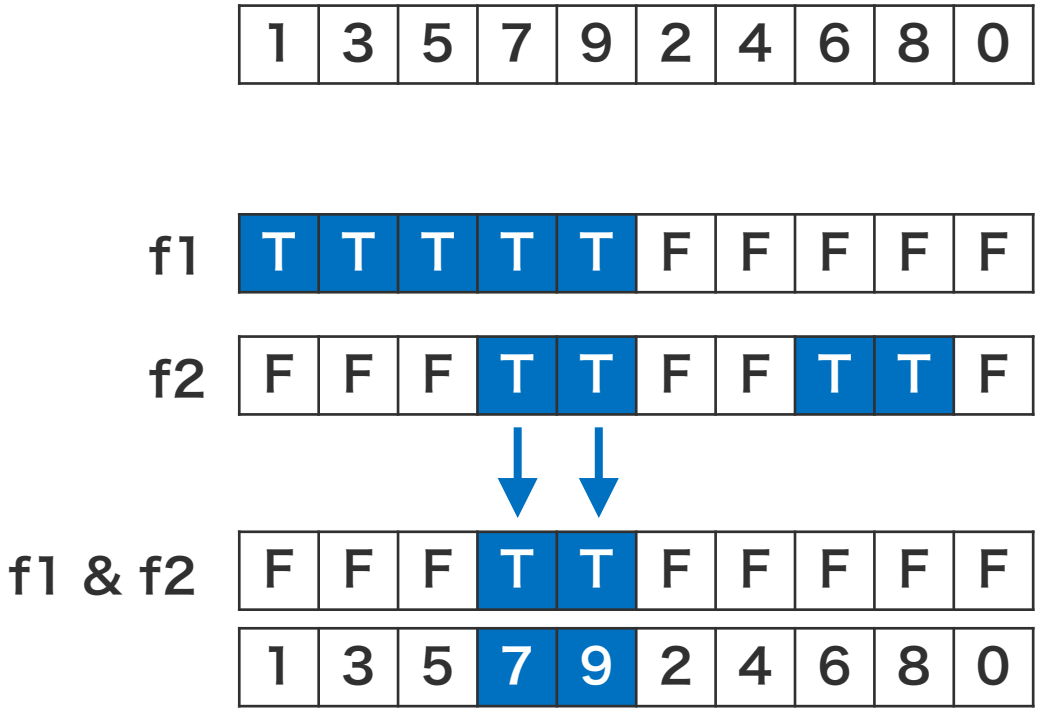

```
import numpy as np

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



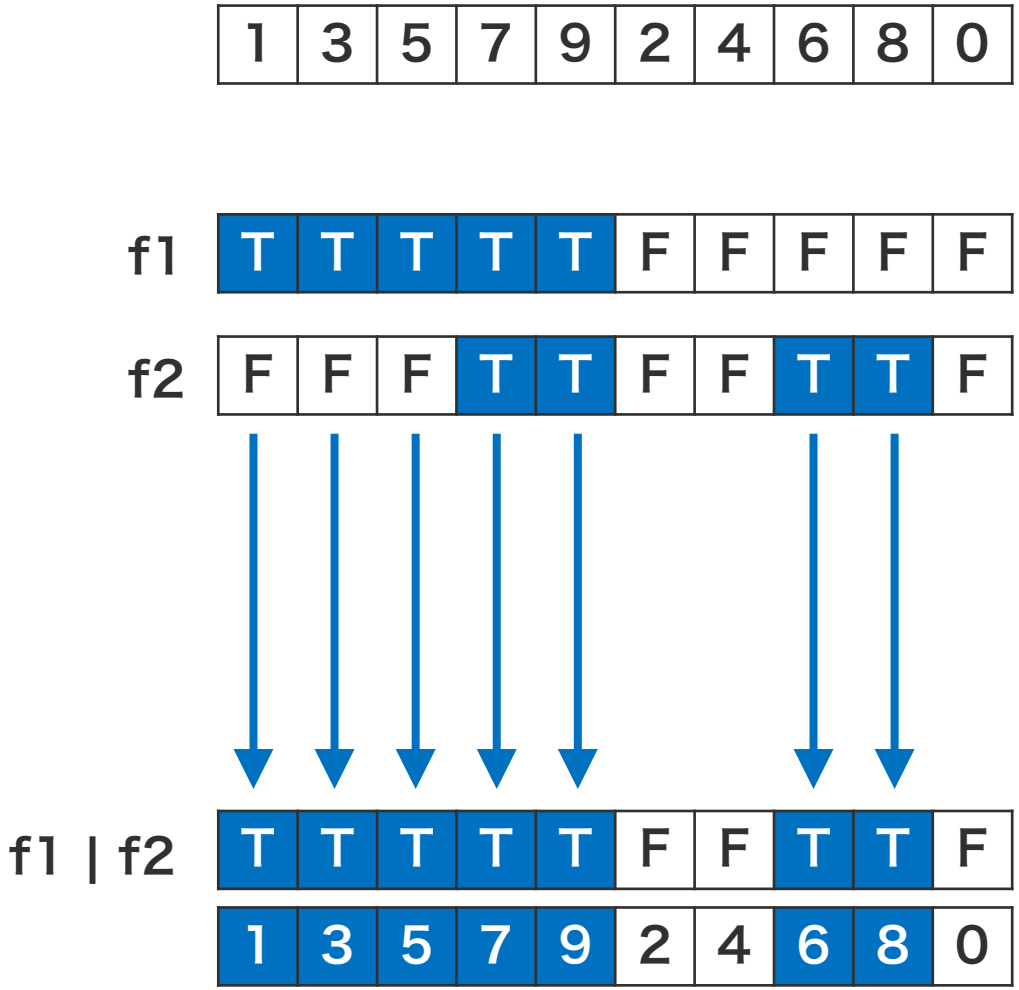

```
import numpy as np

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


f1 = (a % 2 == 1)

f2 = (a > 5)




a[f1 & f2]




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])
```

擬似乱数生成

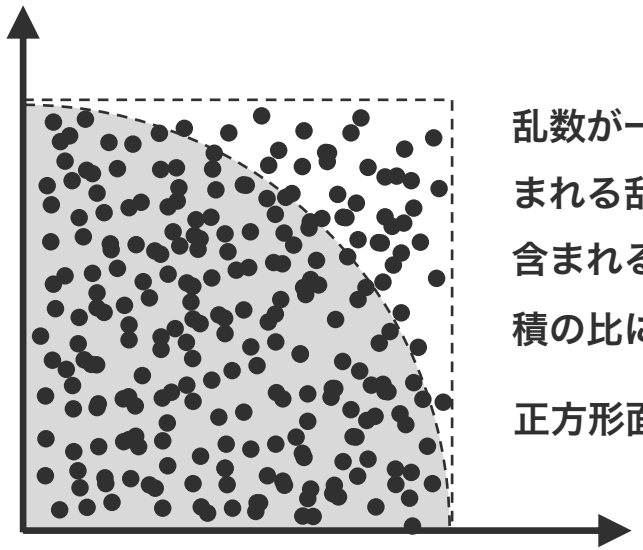
NumPy の random モジュールには擬似乱数を生成する機能が実装されている。次の表に、一様分布および正規分布から乱数を生成する関数を示した。これ以外にも、ポアソン分布やガンマ分布などの分布から乱数を生成する関数が多数用意されている。必要なときに調べて使うとよい。

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

```
import numpy as np

np.random.seed(2020)


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

np.random.rand(n) 関数は、0≦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≦x<1 および 0≦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。

正方形面積：単位円灰色部分 = 1 : $\pi/4$

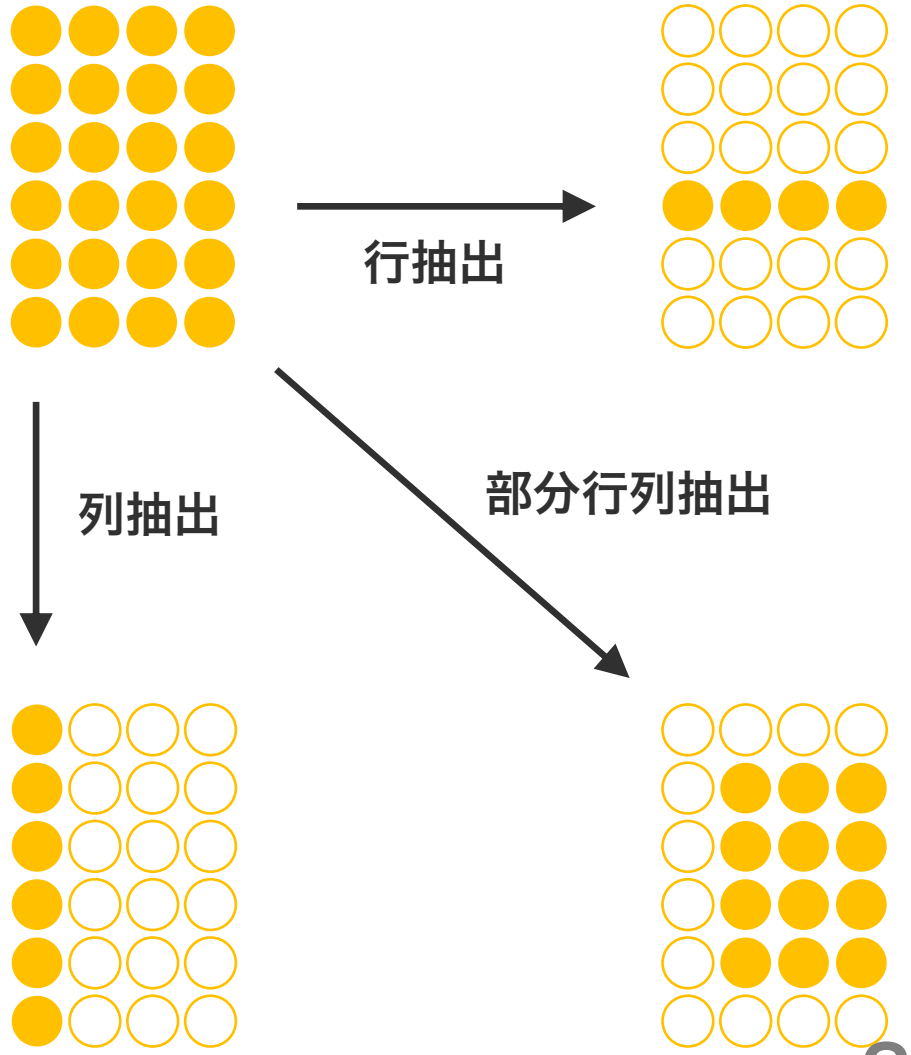
```
import numpy as np

n = 10000
x = np.random.rand(n)
y = np.random.rand(n)
```

NumPy

- １ 次元配列
- ２ 次元配列
- ３ 次元配列
- データの読み書き

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

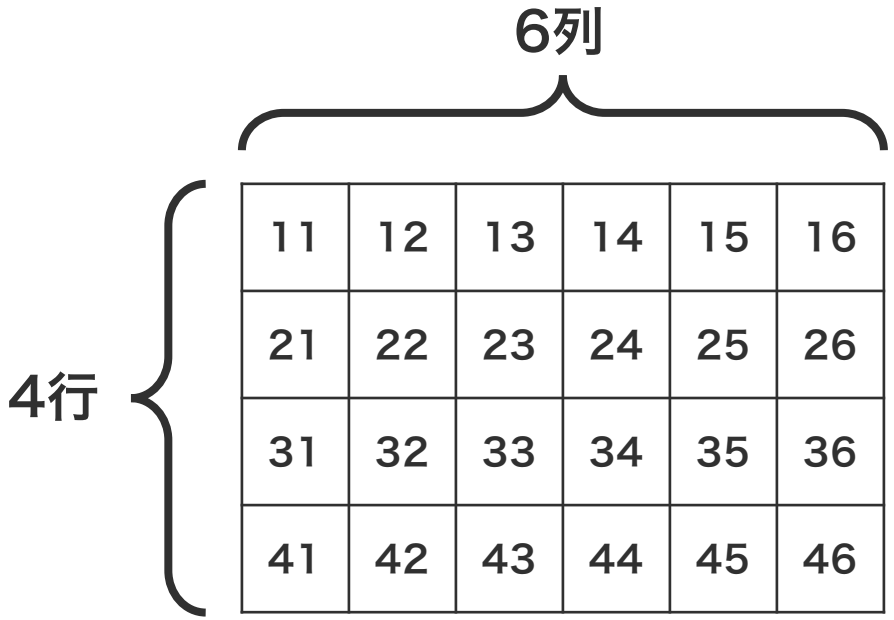
２次元配列の場合、np.identity 関数を使用して、単位行列を作成することができる。

```
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。



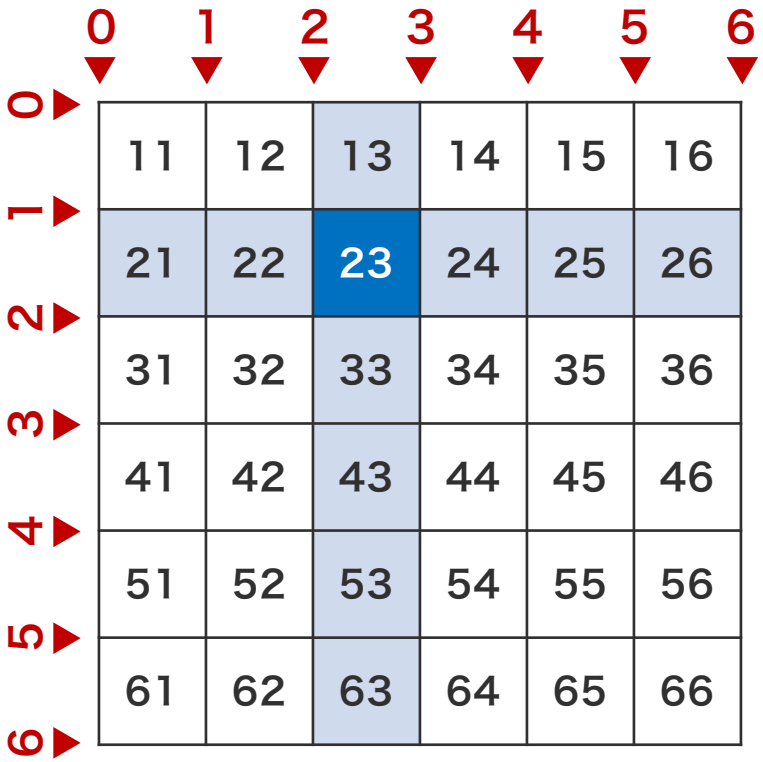

```
a = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46]])


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。

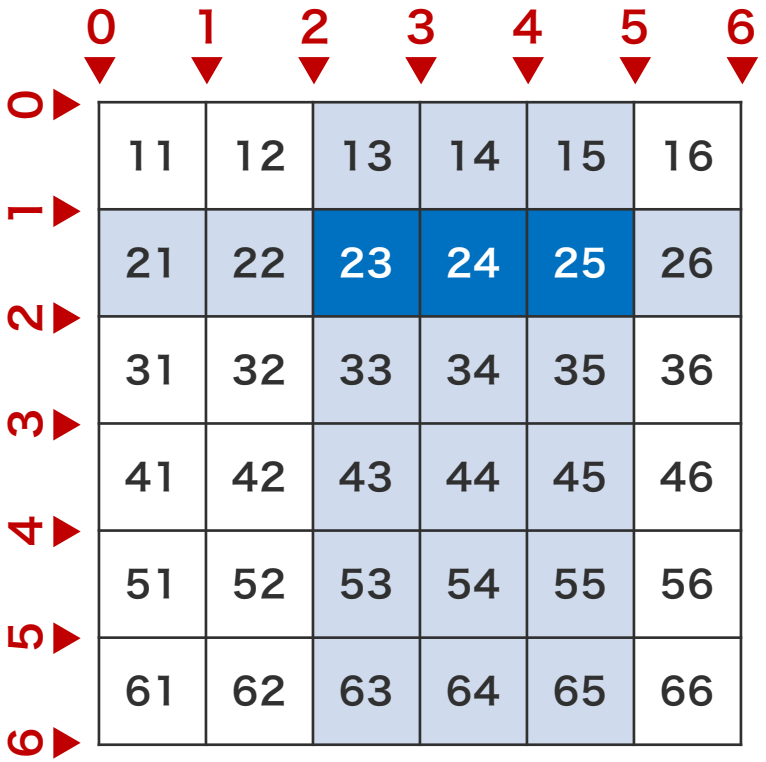
```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])

b[1, 2]


b[1, 2:5]


b[1:5, 3]


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

2 次元配列



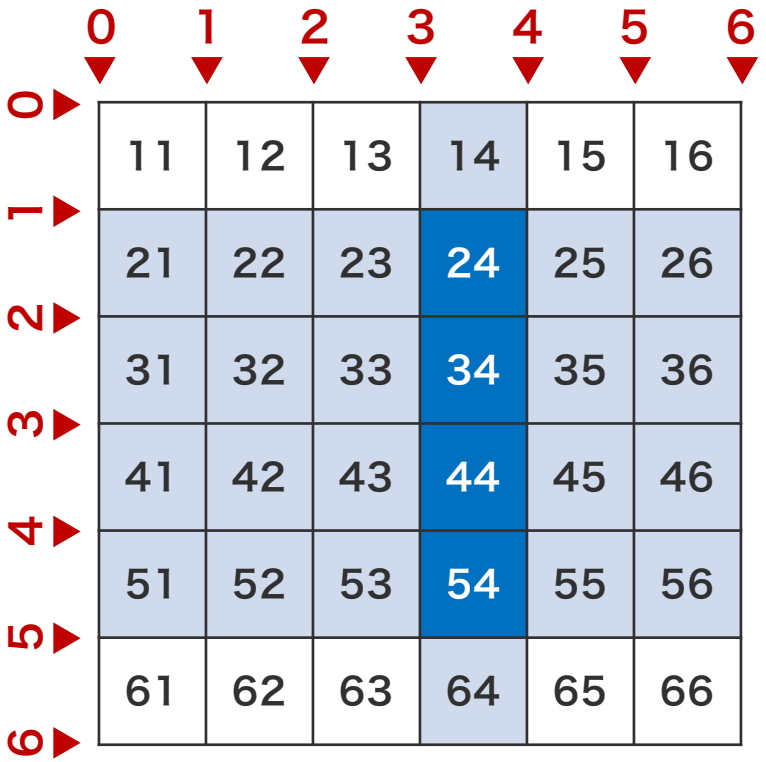

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])

b[1, 2]
# 23

b[1, 2:5]


b[1:5, 3]


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

2 次元配列



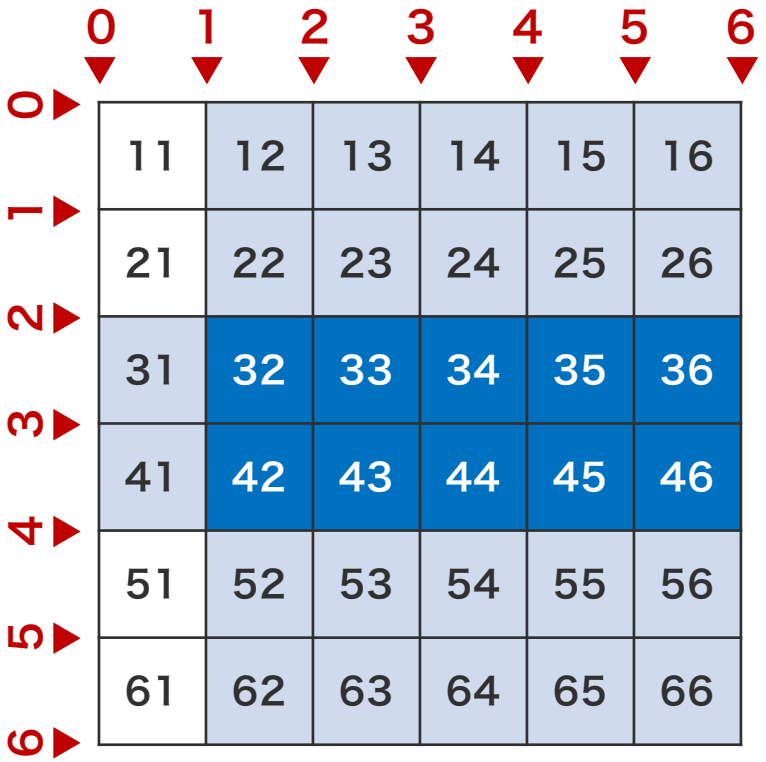

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])

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



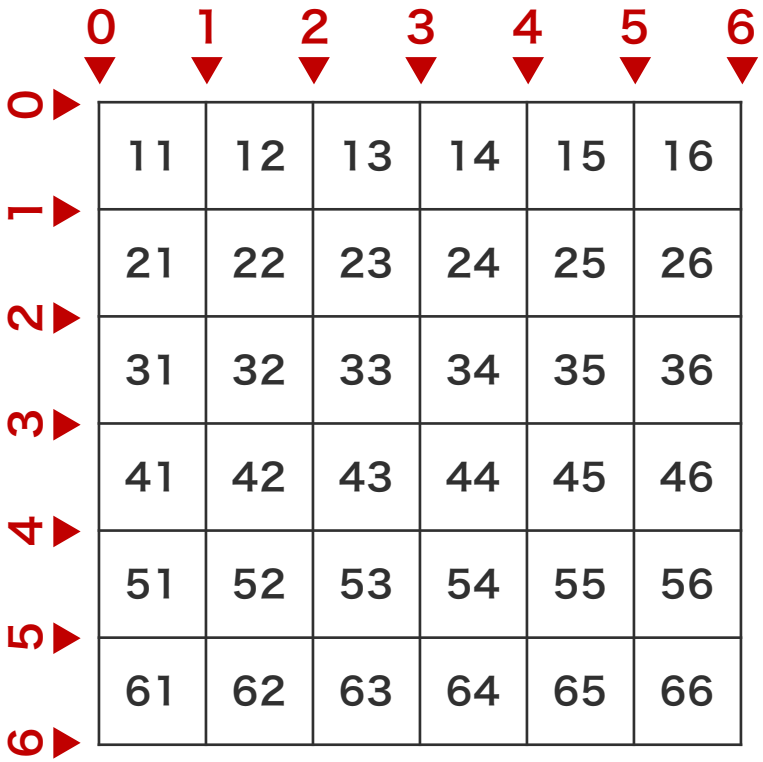

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])

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



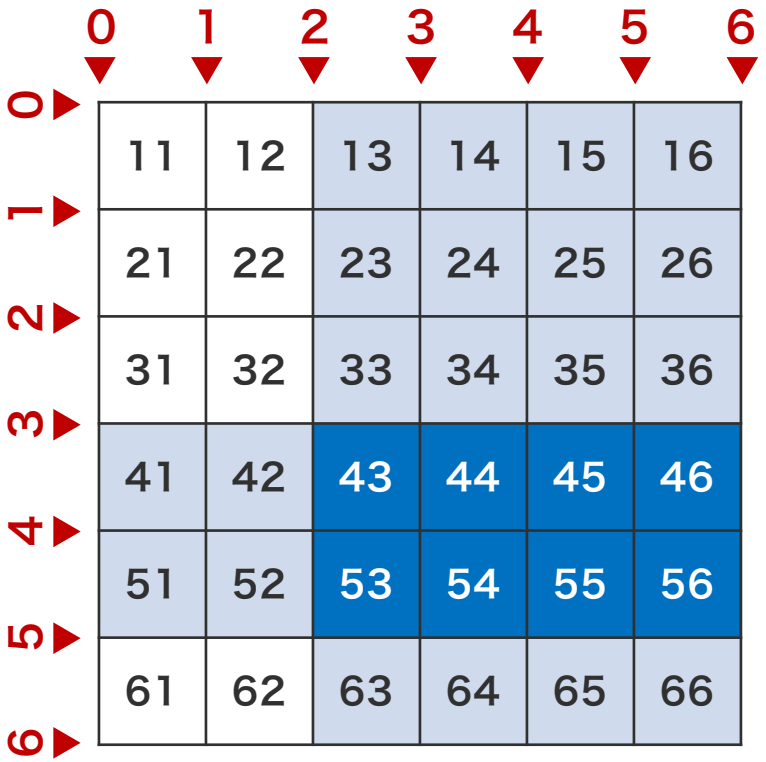

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])

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



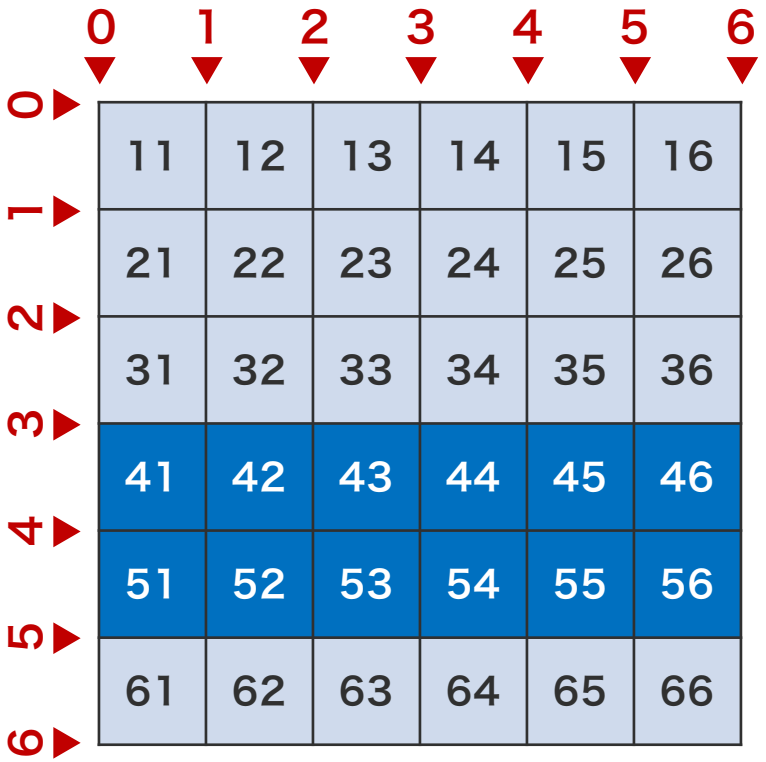

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])

b[3:5, 2:]


b[3:5, :]


b[:, 2]


b[:, 1:3]
```

2 次元配列



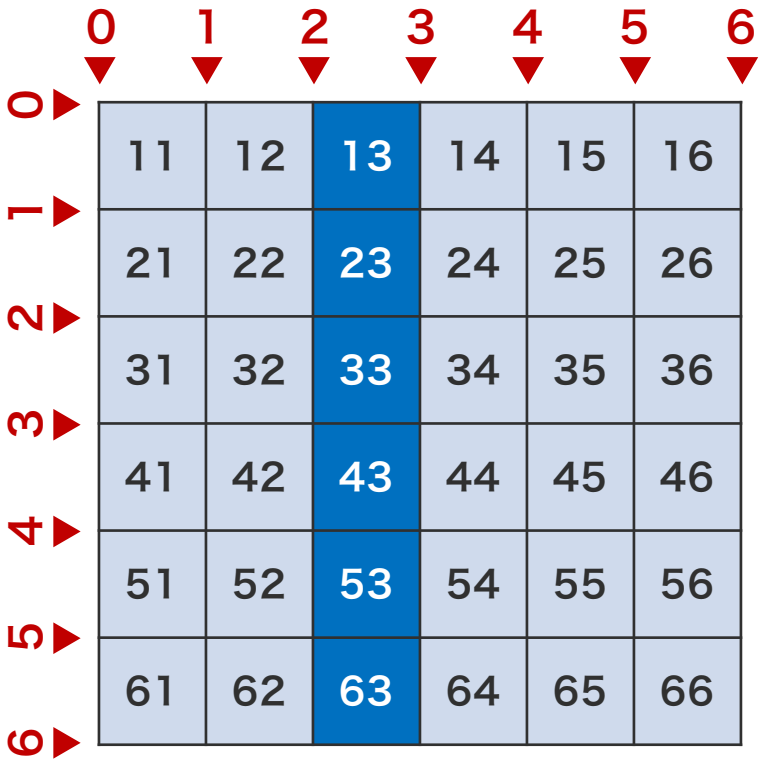

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])

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



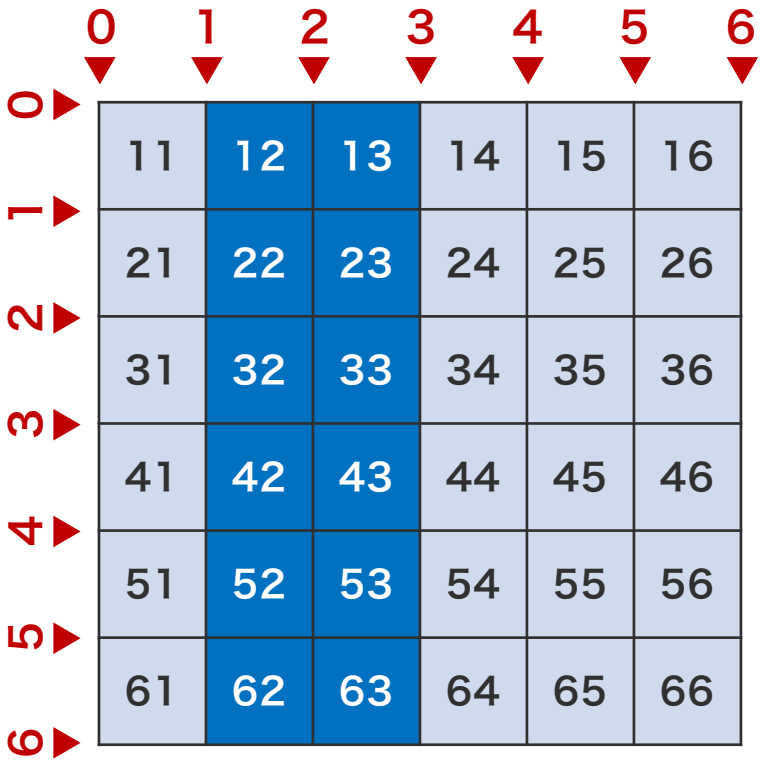

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])

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



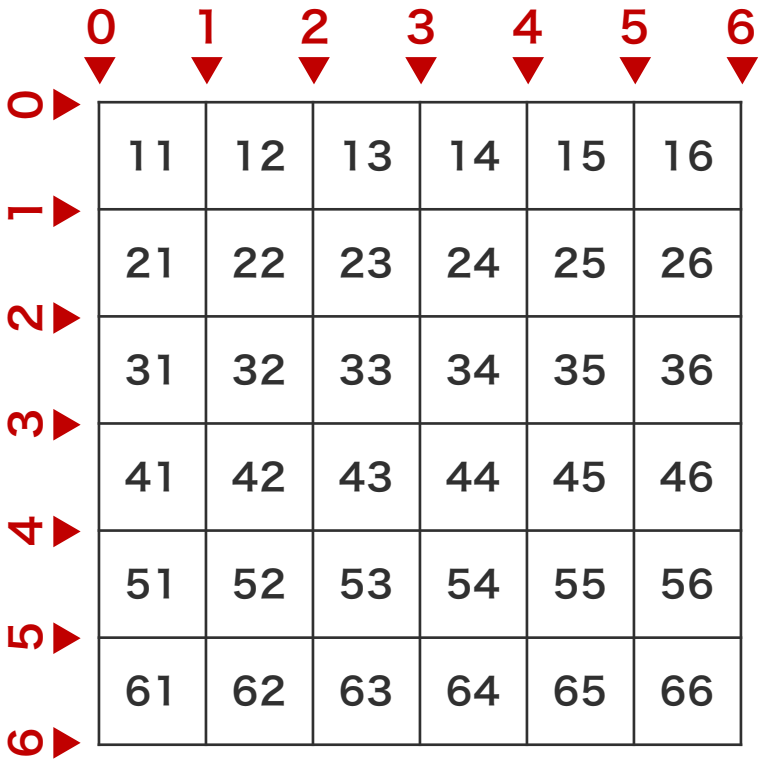

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])

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



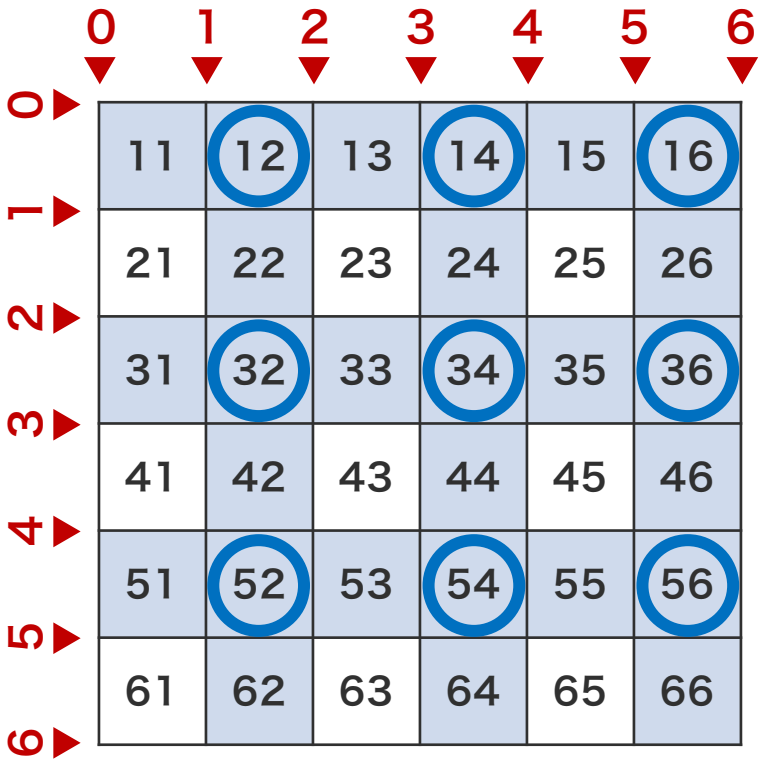

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])

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



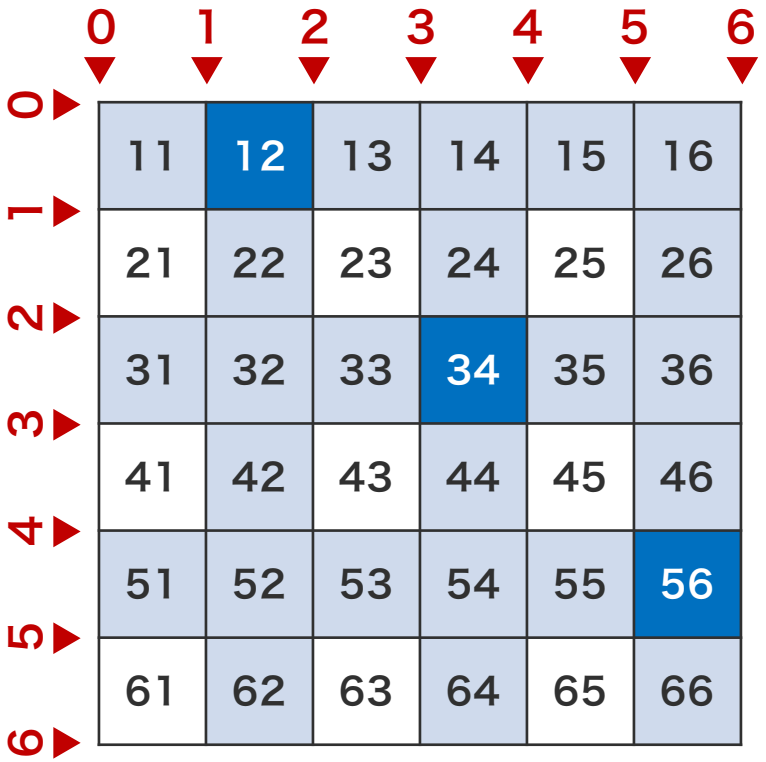

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])

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

b[r, c]
```

2 次元配列 / 注意点



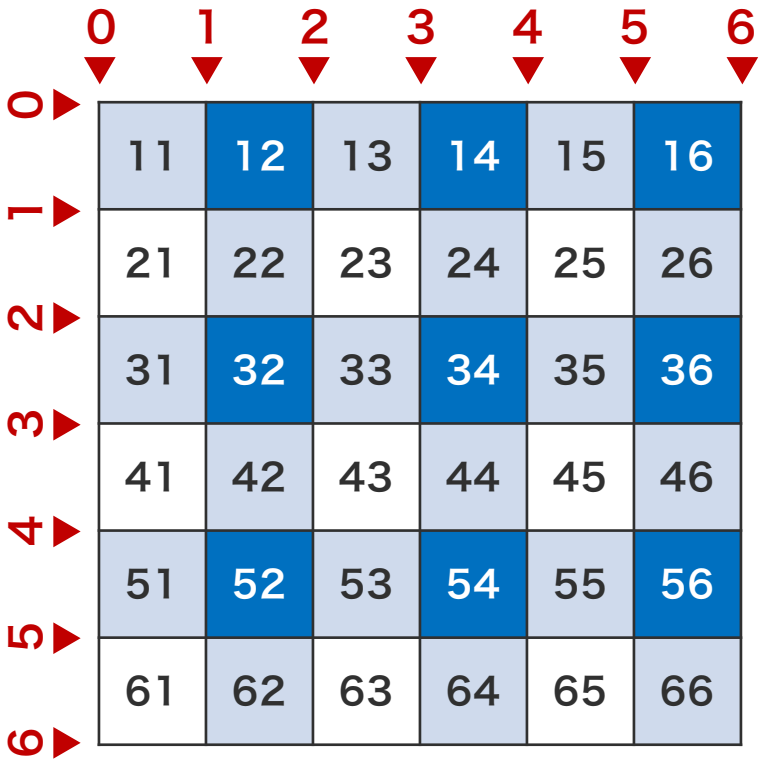

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65, 66]])

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

b[r, c]
```

2 次元配列 / 注意点




```
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




```
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
`np.dot(a, b)`	行列同士の内積
`np.outer(a, b)`	行列同士の外積（テンソル積）
`np.sum(a)`	全要素の和
`a.T`	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

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

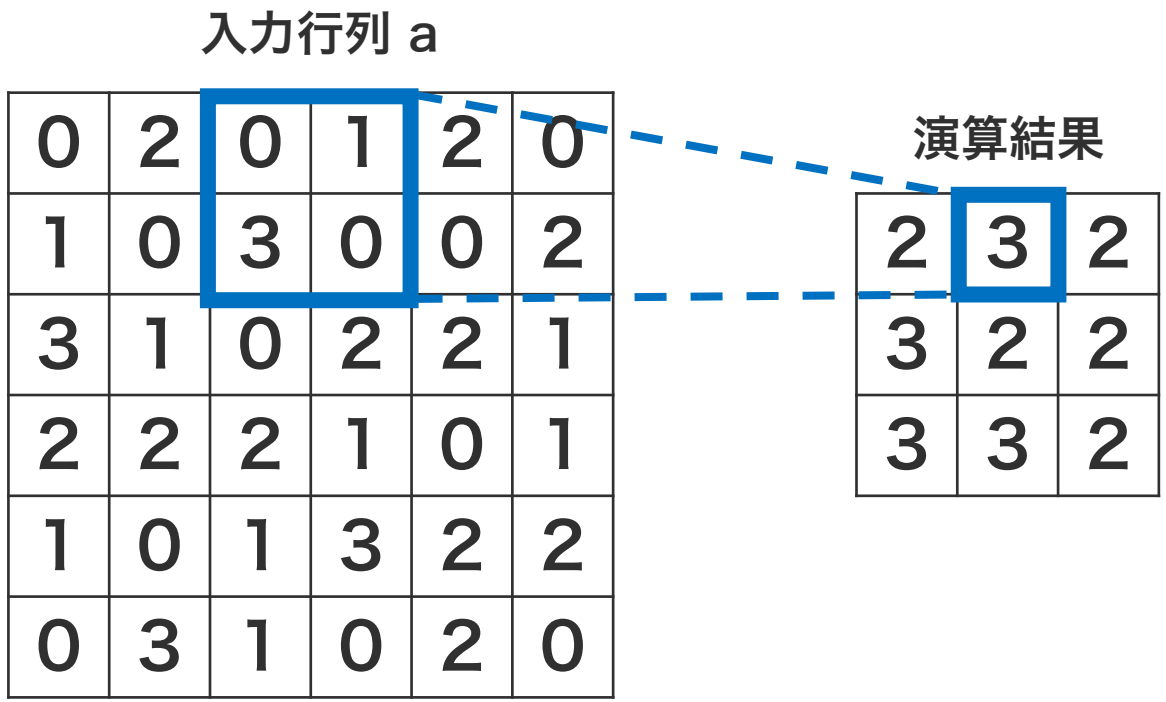
演算結果

2	3	2
3	2	2
3	3	2

問題 N2-1

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。

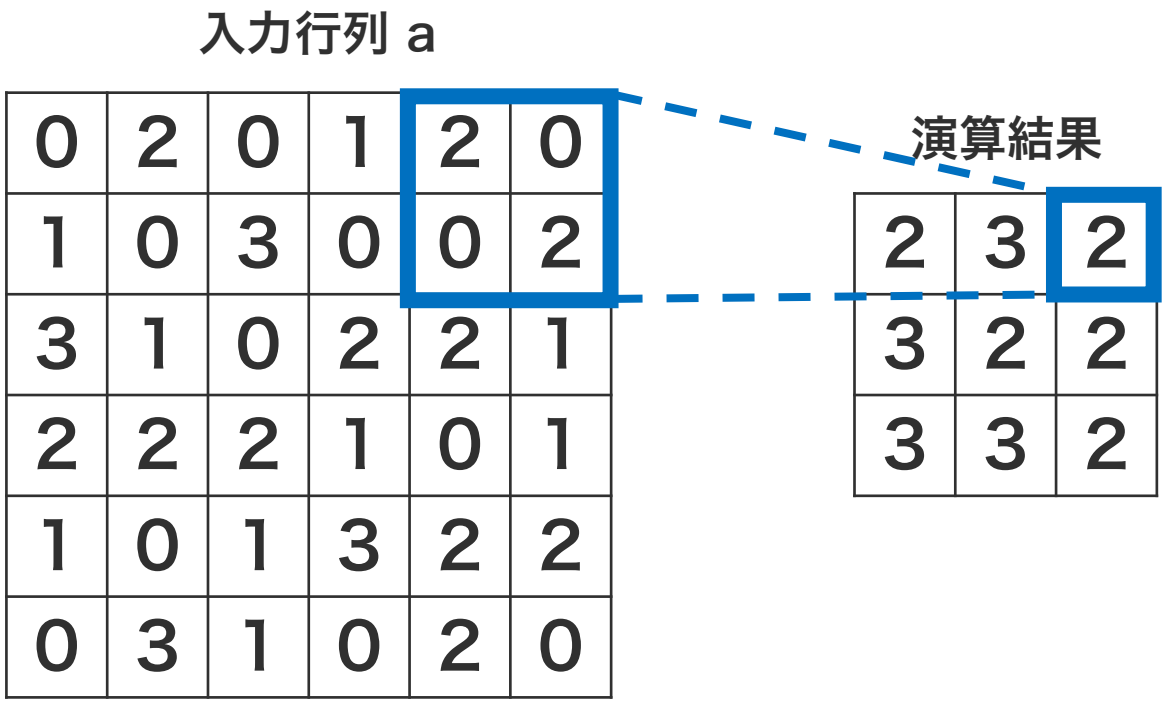
```
a = np.array([[0, 2, 0, 1, 2, 0],
              [1, 0, 3, 0, 0, 2],
              [3, 1, 0, 2, 2, 1],
              [2, 2, 2, 1, 0, 1],
              [1, 0, 1, 3, 2, 2],
              [0, 3, 1, 0, 2, 0]])
```



問題 N2-1

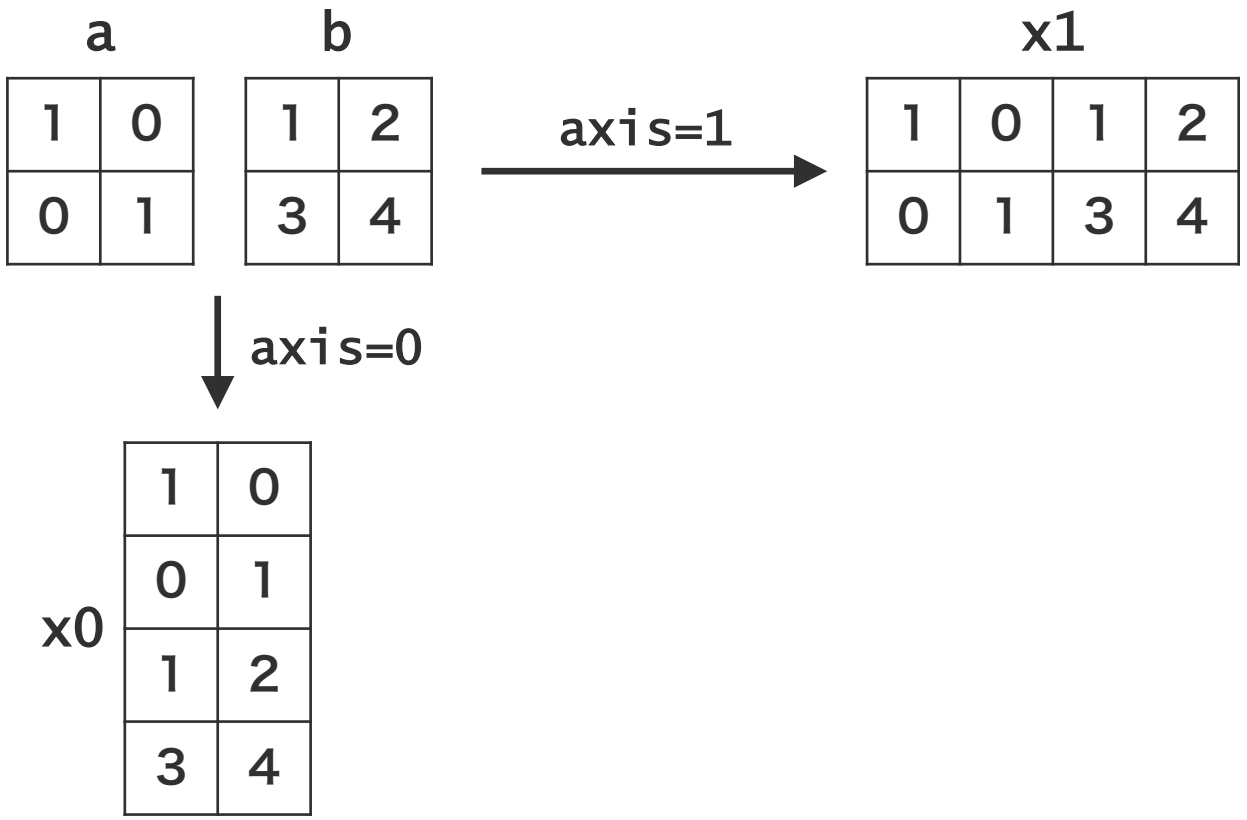
行列 a に対して 2x2 範囲の最大プーリング演算を行い、
その結果を出力せよ。

```
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]])
```



行列操作 / np.concatenate

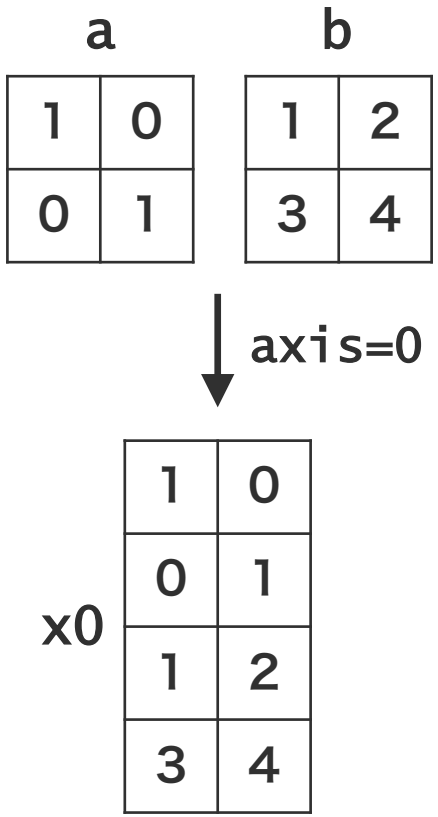
np.concatenate 関数は、複数のNumPy 配列を結合する機能を持つ。この関数を使うとき、結合する軸（次元）方向を axis 引数で指定。省略された場合は axis=0 となる。




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

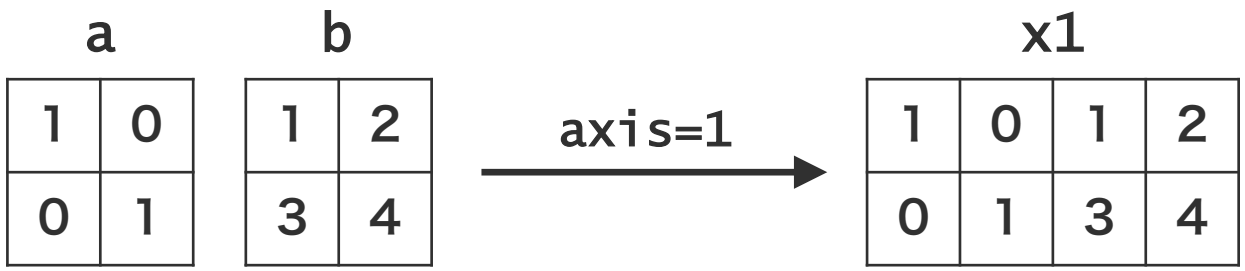
np.vstack 関数は複数の配列を第 1 次元方向に結合する機能を持つ。axis=0 のときの np.concatenate 関数の機能と同じ。




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

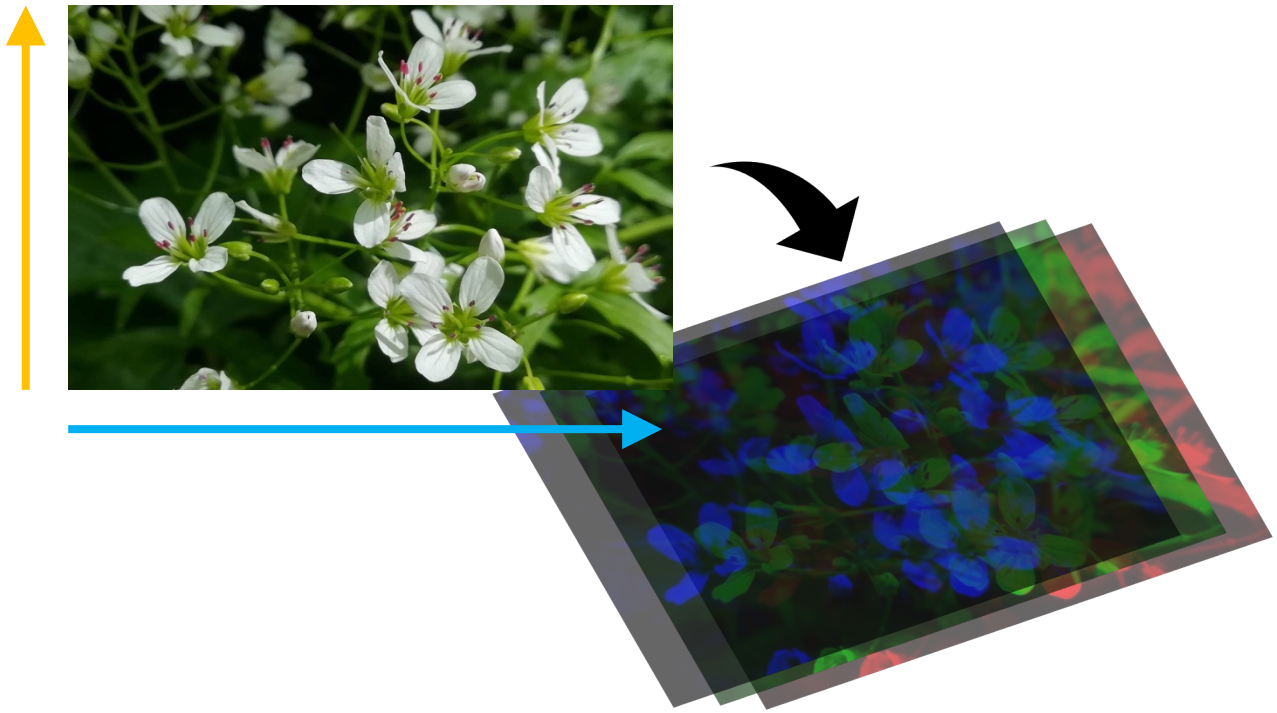
a は 12 要素。最初の 2 次元に 2 要素ずつを配置した場合、最後の次元は自動的に 3 になる。

NumPy

- １ 次元配列
- ２ 次元配列
- ３ 次元配列
- データの読み書き

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 x 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]]])
```

NumPy

- １ 次元配列
- ２ 次元配列
- ３ 次元配列
- データの読み書き

データの書き出し / バイナリー

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。

```
# 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)
```

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

.npy

データの読み込み / バイナリー

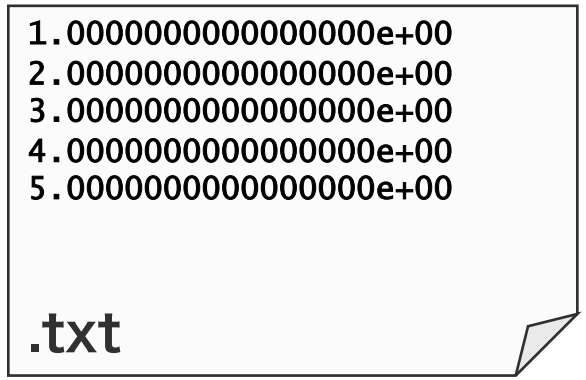
NumPy のバイナリデータを保存しているファイルからデータを読み込むときに load 関数を使用する。複数の配列を含むファイルを読み込むときも load 関数を使用するが、その際、複数個の配列はディクショナリに似たデータ構造でオブジェクトに代入される。そのオブジェクトの .files 属性には、配列の名前の一覧が保存されている。

```
# 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。

```
# 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、genefromtxt 関数を使用する。

```
# text (.txt)

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

農学生命情報科学特論Ⅰ

3

- NumPy
- Pandas

python™

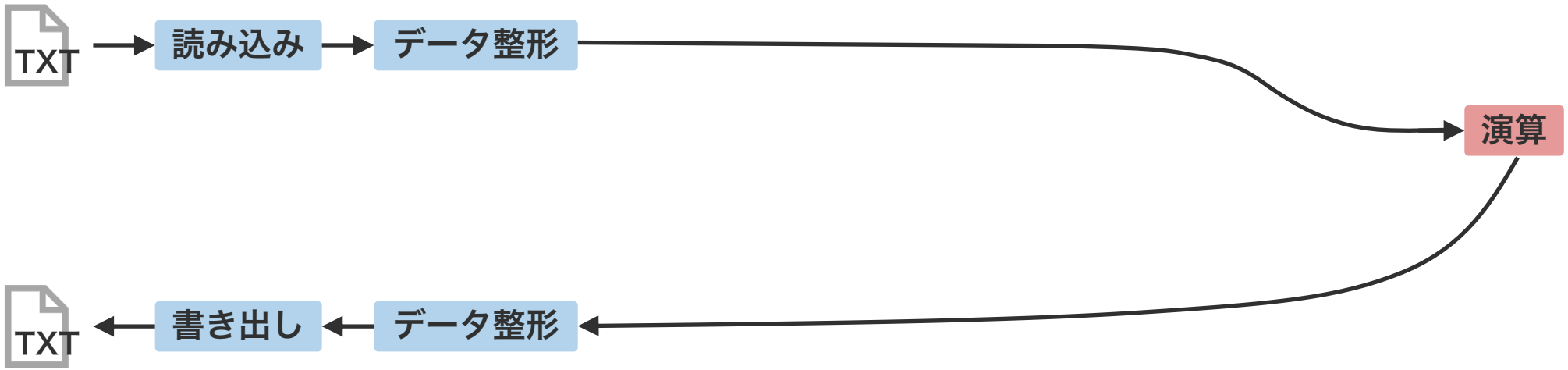
アプリ開発やシステム管理など様々な用途で利用できるような汎用機能を提供する。

- リスト
- 2 次元リスト

NumPy

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



python™

pandas

NumPy

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。

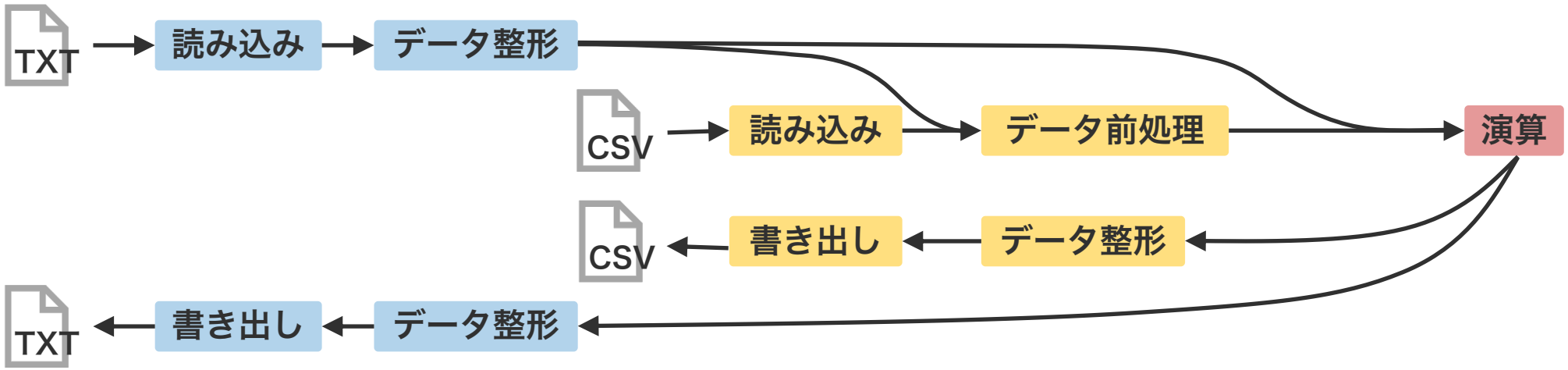
- リスト
- 2 次元リスト

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

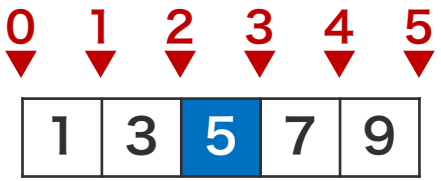


Pandas

- ◯ シリーズ
- ◯ データフレーム
- ◯ 表データ処理

シリーズ

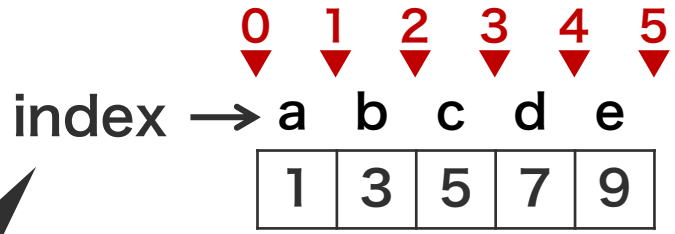
Pandas のシリーズは、1 次元の配列データを扱うときに使用する。pd.Series 関数にリストを代入して作成する。シリーズから要素を取り出すときは、位置番号を指定して取り出す。




```
import pandas as pd

x = pd.Series([1, 3, 5, 7, 9])



x[2]
# 5
```

シリーズ

シリーズは、リストと異なり、各値を位置番号と index の両方で管理している。シリーズを pd.Series 関数で作成するときに、index が自動的に作られるが、自ら指定して作ることもできる。

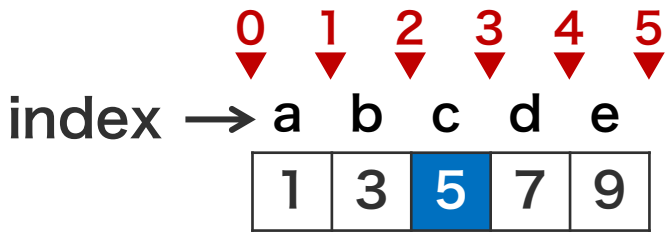


シリーズの各要素に位置番号の他に index と呼ばれる索引が付けられる。

```
import pandas as pd

x = pd.Series([1, 3, 5, 7, 9])
x
# 0    1
# 1    3
# 2    5
# 3    7
# 4    9
# dtype: int64

x = pd.Series([1, 3, 5, 7, 9],
              index=['a', 'b', 'c', 'd', 'e'])
x
# a    1
# b    3
# c    5
# d    7
# e    9
# dtype: int64
```

シリーズ

シリーズを作成するとき、文字列を index に指定した場合、その文字列でシリーズの各要素を取得できるようになる。




```
import pandas as pd

x = pd.Series([1, 3, 5, 7, 9],
              index=['a', 'b', 'c', 'd', 'e'])
x
# a    1
# b    3
# c    5
# d    7
# e    9
# dtype: int64


x[2]
# 5


x['c']
# 5
```

シリーズ

シリーズは、値と index の両方のデータを保持している。シリーズから値だけを NumPy の 1 次元配列として取得したい場合は、x.values のように取得する。また、シリーズから index だけを取得したい場合は、x.index を使用する。

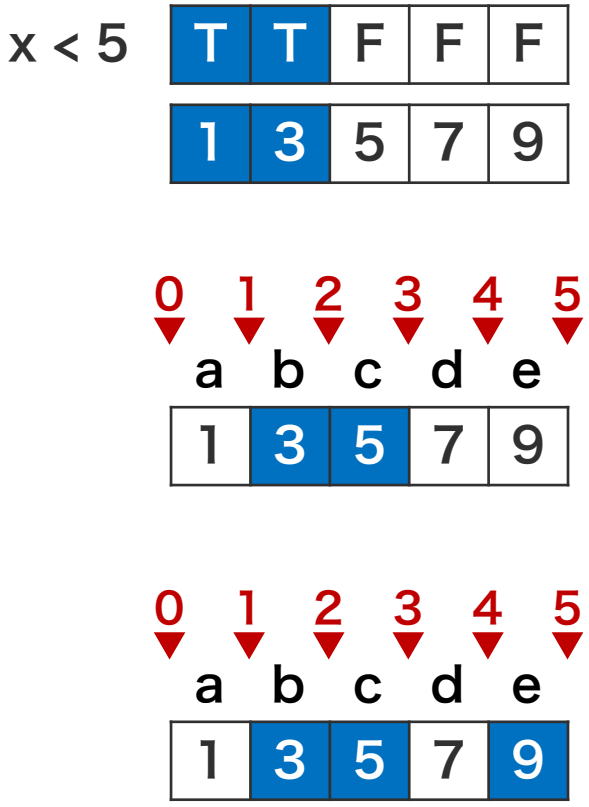
```
import pandas as pd

x = pd.Series([1, 3, 5, 7, 9],
      index=['a', 'b', 'c', 'd', 'e'])

x
# a    1
# b    3
# c    5
# d    7
# e    9
# dtype: int64

x.values
# array([1, 3, 5, 7, 9])

x.index
# Index(['a', 'b', 'c', 'd', 'e'],
#       dtype='object')
```

シリーズ

シリーズから要素を取得するとき、位置番号と index で取得できるほか、NumPy のようにスライスしたり、フィルター（ブーリアンベクトル）を使用して取得したりすることもできる。

```
import pandas as pd
x = pd.Series([1, 3, 5, 7, 9],
              index=['a', 'b', 'c', 'd', 'e'])

x[x < 5]




x[1:3]




k = ['b', 'c', 'e']
x[k]
```

シリーズ




```
import pandas as pd
x = pd.Series([1, 3, 5, 7, 9],
              index=['a', 'b', 'c', 'd', 'e'])

x[x < 5]
# a    1
# b    3
# dtype: int64

x[1:3]
# b    3
# c    5
# dtype: int64

k = ['b', 'c', 'e']
x[k]
# b    3
# c    5
# e    9
# dtype: int64
```

問題 P1-1

赤色で書かれているオブジェクトが保持している値を答えよ。

```
import pandas as pd
x = pd.Series([1, 3, 5, 7, 9],
        index=['a', 'b', 'c', 'd', 'e'])


x[['a', 'c', 'e']]




keep1 = (1 < x)
keep2 = (x < 7)
x[keep1 & keep2]
```

```
import pandas as pd
x = pd.Series([1, 3, 5, 7, 9],
        index=['a', 'b', 'c', 'd', 'e'])

x['a'] = 2
x[0] = 4
x.values


x[x > 4] = 6
x.values


x[x % 2 == 0] = 1
x.values
```

シリーズ同士の計算

シリーズ同士は NumPy の配列と同様に四則演算が可能である。計算対象のシリーズの長さが同一でない場合は、見かけ上、短い方に欠損値を埋め込んだ上で計算される。また、シリーズとスカラーの計算も定義されている。この場合、スカラーが自動的に、シリーズと同じ長さに展開されて、計算が行われる（ブロードキャスト）。

```
import pandas as pd

x = pd.Series([1, 3, 5, 9])
y = pd.Series([2, 4, 6, 8])
z = pd.Series([1, 1])
w = 2

a = x + y
a.values
#array([ 3,  7, 11, 17])


b = x - z
b.values
# array([ 0.,  2., nan, nan])

c = x * w
c.values
# array([ 2,  6, 10, 18])
```

シリーズ同士の計算

シリーズに index がついている場合、計算は index に基づいて計算される。どちらか一方のシリーズにしか存在しない index の場合、存在しない方を欠損値として扱う。

```
import pandas as pd


x = pd.Series([1, 3, 5, 7],
              index=['a', 'b', 'c', 'd'])
y = pd.Series([2, 4, 6, 8],
              index=['d', 'c', 'b', 'a'])
a = x + y
a.values
# array([9, 9, 9, 9])


x = pd.Series([1, 3, 5, 7],
              index=['a', 'b', 'c', 'd'])
y = pd.Series([2, 4, 6, 8],
              index=['a', 'b', 'd', 'c'])
a = x * y
a.values
# array([ 2,  12, 40, 42)
```

計算後の結果は index 順に並べ替えられる。x * y と y * x の計算結果は同じである。

シリーズ同士の計算

シリーズに index がついている場合、計算は index に基づいて計算される。どちらか一方のシリーズにしか存在しない index の場合、存在しない方を欠損値として扱う。

```
import pandas as pd


x = pd.Series([1, 3, 5, 7],
              index=['a', 'b', 'd', 'e'])

y = pd.Series([2, 4, 6, 8],
              index=['a', 'b', 'c', 'e'])


a = x + y
a.values
# array([ 3.,  7., nan, nan, 15.])


b = x * y
b.values
# array([ 2., 12., nan, nan, 56.])
```

要約統計量

Pandas のシリーズに対して、平均、分散、中央値などの要約統計量を計算するメソッドが多く用意されている。

```
import numpy as np
import pandas as pd
x = pd.Series([1, 2, 3, 4, 5])

x.count()
# 5

x.min()
# 1

x.max()
# 5

x.idxmax()
# 4

x.quantile(0.25)
# 2.0
```

```
x.sum()
# 15

x.mean()
# 3.0

x.median()
# 3.0

x.var()
# 2.5

x.std()
# 1.5811388300841898

x.cumsum().values
# array([ 1,  3,  6, 10, 15])
```

欠損値 / dropna

Pandas のシリーズに欠損値・非数値（np.nan）を含めることができる。欠損値は、Pandas で用意されたメソッドを使って取り除いたり、その位置を調べたりすることができる。

メソッド	動作
dropna	シリーズ中の np.nan を取り除く。
fillna	シリーズ中の np.nan を指定した値で置き換える。
isnull	シリーズ中の各要素が np.nan かどうかを True と False で表す。
notnull	is.null と反対の動作を行う。

欠損値除去後のシリーズの要素数が変化するので、複数のシリーズを同時に解析するとき、要素数の違いによりブロードキャスト機能が働き、想定外の計算が行われる可能があることに注意。

```
import numpy as np
import pandas as pd

x = pd.Series([1, 3, np.nan, 7, np.nan])
x
# a      1.0
# b      3.0
# c      NaN
# d      7.0
# e      NaN
# dtype: float64

y = x.dropna()
y
# 0 1.0
# 1 3.0
# 3 7.0
# dtype: float64
```

欠損値 / fillna

Pandas のシリーズに欠損値・非数値（np.nan）を含めることができる。欠損値は、Pandas で用意されたメソッドを使って取り除いたり、その位置を調べたりすることができる。

メソッド	動作
dropna	シリーズ中の np.nan を取り除く。
fillna	シリーズ中の np.nan を指定した値で置き換える。
isnull	シリーズ中の各要素が np.nan かどうかを True と False で表す。
notnull	is.null と反対の動作を行う。

合理的な根拠なしに、すべての欠損値を特定の値に置き換えてはならない。fillna メソッドを使用するときは十分に注意すること。

```
import numpy as np
import pandas as pd

x = pd.Series([1, 3, np.nan, 7, np.nan])


y = x.fillna(0)
y
# 0 1.0
# 1 3.0
# 2 0.0
# 3 7.0
# 4 0.0
# dtype: float64
```

欠損値 / isnull

Pandas のシリーズに欠損値・非数値（np.nan）を含めることができる。欠損値は、Pandas で用意されたメソッドを使って取り除いたり、その位置を調べたりすることができる。

メソッド	動作
dropna	シリーズ中の np.nan を取り除く。
fillna	シリーズ中の np.nan を指定した値で置き換える。
isnull	シリーズ中の各要素が np.nan かどうかを True と False で表す。
notnull	is.null と反対の動作を行う。

```
import numpy as np
import pandas as pd

x = pd.Series([1, 3, np.nan, 7, np.nan])


y = x.isnull()
y
# 0 False
# 1 False
# 2 True
# 3 False
# 4 True
# dtype: bool
```

欠損値 / notnull

Pandas のシリーズに欠損値・非数値（np.nan）を含めることができる。欠損値は、Pandas で用意されたメソッドを使って取り除いたり、その位置を調べたりすることができる。

メソッド	動作
dropna	シリーズ中の np.nan を取り除く。
fillna	シリーズ中の np.nan を指定した値で置き換える。
isnull	シリーズ中の各要素が np.nan かどうかを True と False で表す。
notnull	is.null と反対の動作を行う。

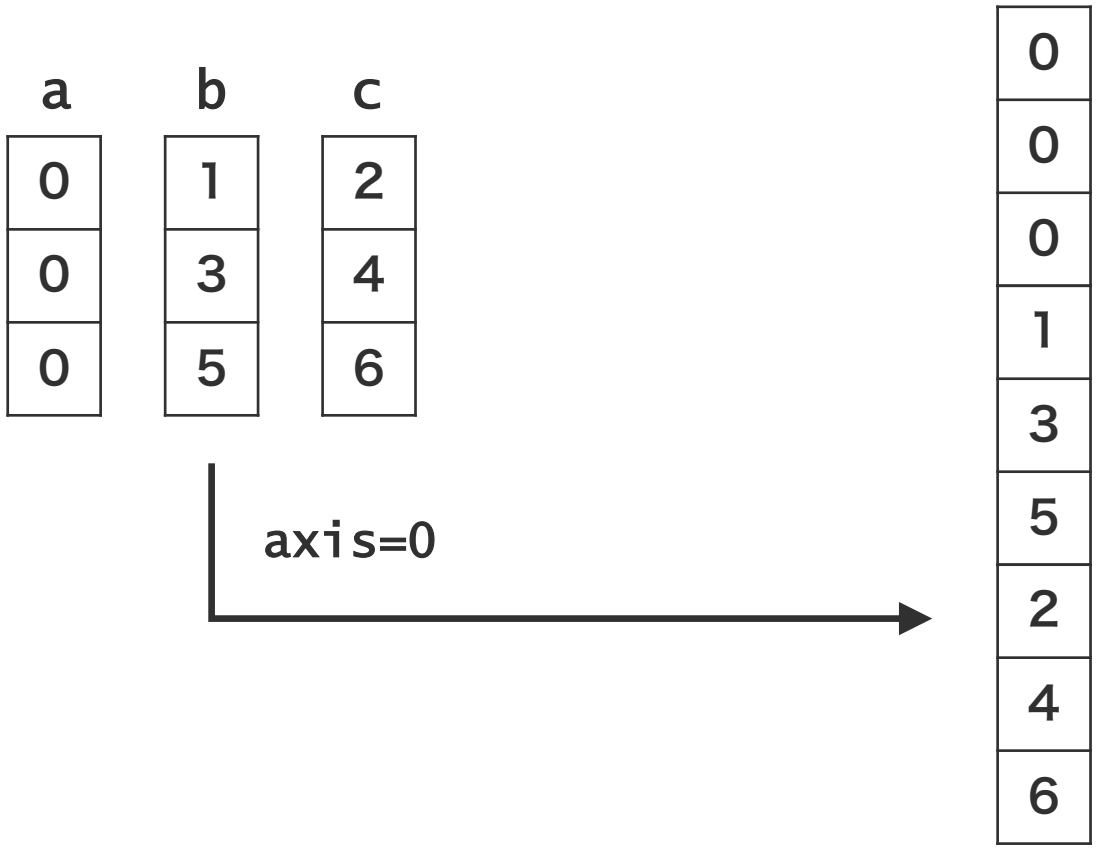
```
import numpy as np
import pandas as pd

x = pd.Series([1, 3, np.nan, 7, np.nan])


y = x.notnull()
y
# 0 True
# 1 True
# 2 False
# 3 True
# 4 False
# dtype: bool
```

Pandas

- シリーズ
- データフレーム
- 表データ処理

データフレーム / pd.concat

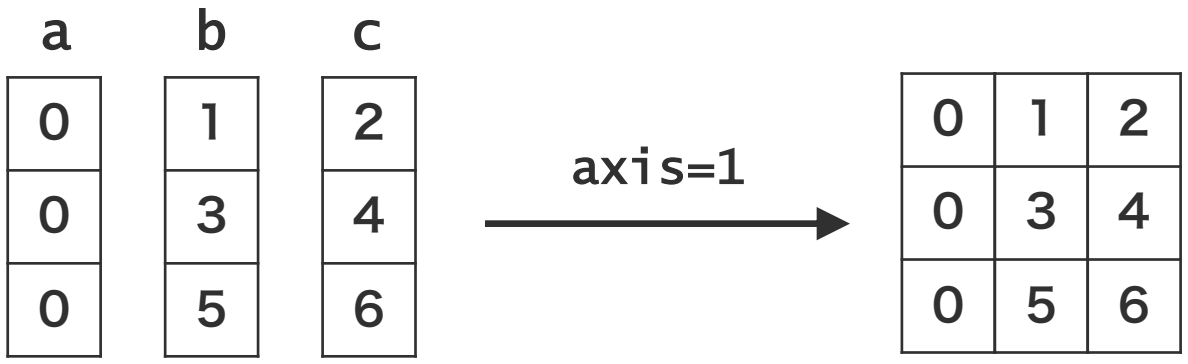
Pandas では、行列型のデータをデータフレームと呼ぶ。データフレームは、シリーズを行方向あるいは列方向に束ねることで作成される。シリーズ同士を束ねるとき pd.concat 関数を使用する。




```
import pandas as pd
a = pd.Series([0, 0, 0])
b = pd.Series([1, 3, 5])
c = pd.Series([2, 4, 6])

x = pd.concat([a, b, c])
x
# 0    0
# 1    0
# 2    0
# 0    1
# 1    3
# 2    5
# 0    2
# 1    4
# 2    6
# dtype: int64

x.__class__.__name__
# 'Series'
```

データフレーム / pd.concat

pd.concat 関数の axis 引数を指定することで、複数のシリーズを列方向に束ねることもできる。




```
import pandas as pd
a = pd.Series([0, 0, 0])
b = pd.Series([1, 3, 5])
c = pd.Series([2, 4, 6])

y = pd.concat([a, b, c], axis=1)
y
#    0  1  2
# 0  0  1  2
# 1  0  3  4
# 2  0  5  6

y.__class__.__name__
# 'DataFrame'
```

データフレーム / ディクショナリ

リストを値として保存しているディクショナリを、Pandas のデータフレームに変換することもできる。このとき、ディクショナリのキーは、データフレームの列名に変換される。

```
import pandas as pd


d = {'buna' : [1, 0, 1, 0, 1],
     'kashi': [1, 3, 5, 7, 9],
     'nara' : [0, 2, 4, 6, 8]}


df = pd.DataFrame(d)
df
#    buna kashi  nara
# 0     1     1     0
# 1     0     3     2
# 2     1     5     4
# 3     0     7     6
# 4     1     9     8
```

データフレーム / ディクショナリ

シリーズを値として保存しているディクショナリも Pandas のデータフレームに変換することもできる。ディクショナリのキーは、データフレームの列名に変換される。また、シリーズの index は、データフレームの index （行名）に変換される。

```
import pandas as pd

w1 = pd.Series([1, 0, 1, 0, 1],
         index=['a', 'b', 'c', 'd', 'e'])
w2 = pd.Series([1, 3, 5, 7, 9],
         index=['a', 'b', 'c', 'd', 'e'])
w3 = pd.Series([0, 2, 4, 6, 8],
         index=['a', 'b', 'c', 'd', 'e'])

d = {'buna': w1, 'kashi': w2,
     'nara': w3}
df = pd.DataFrame(d)
df
#    buna kashi  nara
# a     1     1     0
# b     0     3     2
# c     1     5     4
# d     0     7     6
# e     1     9     8
```

データフレーム / ディクショナリ

index を含むシリーズの場合、データフレームはそれらの index に基づいて作られる。データフレームは index で並べ替えられるため、その順番は必ずしもシリーズの順番を反映しないことに注意。

```
import pandas as pd

w1 = pd.Series([1, 0, 1, 0, 1],
         index=['b', 'a', 'c', 'd', 'f'])
w2 = pd.Series([1, 3, 5, 7, 9],
         index=['a', 'b', 'c', 'f', 'e'])
w3 = pd.Series([0, 2, 4, 6, 8],
         index=['c', 'b', 'a', 'd', 'e'])

d = {'buna': w1, 'kashi': w2,
     'nara': w3}
df = pd.DataFrame(d)
df
#    buna  kashi  nara
# a   0.0    1.0   4.0
# b   1.0    3.0   2.0
# c   1.0    5.0   0.0
# d   0.0    NaN   6.0
# e   NaN    9.0   8.0
# f   1.0    7.0   NaN
```

データフレーム / 二次元リスト

二次元リストを作成し、それをデータフレームに変換することもできる。この際に、データフレームの列名（columns）と行名（index）が自動的に振られる。なお、データフレームを作成するときに、columns と index 引数を利用することで、列名と行名を自由につけることができる。

```
import pandas as pd
d = pd.DataFrame([[11, 12, 13, 14],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]])
d
#    0  1  2  3
# 0 11 12 13 14
# 1 21 22 23 24
# 2 31 32 33 34

d = pd.DataFrame([[11, 12, 13, 14],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]],
        index=['R1', 'R2', 'R3'],
      columns=['C1', 'C2', 'C3', 'C4'])
d
#    C1 C2 C3 C4
# R1 11 12 13 14
# R2 21 22 23 24
# R3 31 32 33 34
```

データフレーム / 行名と列名

データフレームの index と columns は、あとから変更することができる。データフレームの index と columns 属性に直接新しい名前を代入することで変更できる。

```
import pandas as pd

d = pd.DataFrame([[11, 12, 13, 14],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]],
        index=['R1', 'R2', 'R3'],
      columns=['C1', 'C2', 'C3', 'C4'])
d
#     C1  C2  C3  C4
# R1  11  12  13  14
# R2  21  22  23  24
# R3  31  32  33  34

d.index = ['x', 'y', 'z']
d.columns = ['h', 'i', 'j', 'k']
d
#    h  i  j  k
# x 11 12 13 14
# y 21 22 23 24
# z 31 32 33 34
```

特定の列または行の名前だけを変更したいとき、Pandas の rename メソッドを使用すると便利である。

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rename.html

データフレーム要素参照 / iloc

データフレームから要素を取得する方法として、位置番号を利用しても、行名・列名を利用しても取得できる。位置番号で取得する場合は、iloc を使用し、0 から始まる位置番号を与える。

	C1	C2	C3	C4
R1	11	12	13	14
R2	21	22	23	24
R3	31	32	33	34

R1	11	12	13	14
R2	21	22	23	24
R3	31	32	33	34

```
import pandas as pd

d = pd.DataFrame([[11, 12, 13, 14],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]],
        index=['R1', 'R2', 'R3'],
      columns=['C1', 'C2', 'C3', 'C4'])


d.iloc[0, :]
# C1 11
# C2 12
# C3 13
# C4 14

d.iloc[:, 2]
# R1 13
# R2 23
# R3 33
```

データフレーム要素参照 / iloc

	C1	C2	C3	C4
R1	11	12	13	14
R2	21	22	23	24
R3	31	32	33	34

R1	11	12	13	14
R2	21	22	23	24
R3	31	32	33	34

```
import pandas as pd

d = pd.DataFrame([[11, 12, 13, 14],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]],
        index=['R1', 'R2', 'R3'],
      columns=['C1', 'C2', 'C3', 'C4'])


d.iloc[0:2, 1:4]
#     C2 C3 C4
# R1 12 13 14
# R2 22 23 24



d.iloc[[0, 2], [1, 3]]
#     C2 C4
# R1 12 14
# R3 32 34
```

NumPy の動作と異なるので、両者を混同しないように。

データフレーム要素参照 / loc

データフレームから要素を行名または列名で取得するときは、loc を使用する。

	C1	C2	C3	C4
R1	11	12	13	14
R2	21	22	23	24
R3	31	32	33	34

R1	11	12	13	14
R2	21	22	23	24
R3	31	32	33	34

```
import pandas as pd

d = pd.DataFrame([[11, 12, 13, 14],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]],
        index=['R1', 'R2', 'R3'],
      columns=['C1', 'C2', 'C3', 'C4'])


d.loc['R1', :]
# C1 11
# C2 12
# C3 13
# C4 14

d.loc[:, 'C3']
# R1 13
# R2 23
# R3 33
```

データフレーム要素参照 / loc

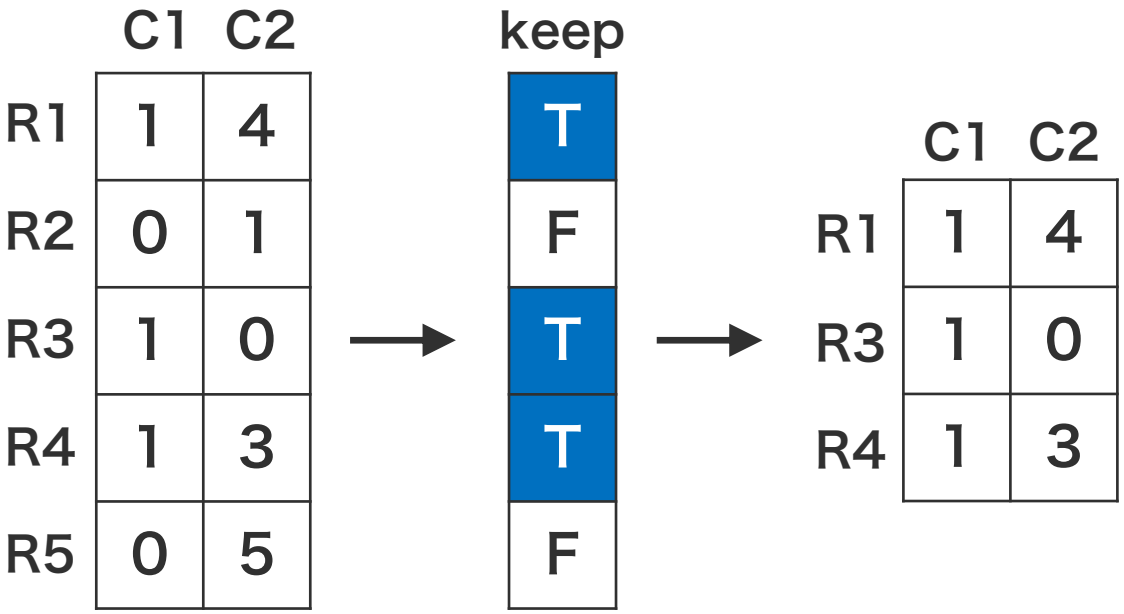
	C1	C2	C3	C4
R1	11	12	13	14
R2	21	22	23	24
R3	31	32	33	34

R1	11	12	13	14
R2	21	22	23	24
R3	31	32	33	34

```
import pandas as pd

d = pd.DataFrame([[11, 12, 13, 14],
                  [21, 22, 23, 24],
                  [31, 32, 33, 34]],
        index=['R1', 'R2', 'R3'],
      columns=['C1', 'C2', 'C3', 'C4'])


d.loc[['R1', 'R2'], 'C2':'C4']
#     C2 C3 C4
# R1 12 13 14
# R2 22 23 24



d.loc[['R1', 'R3'], ['C2', 'C4']]
#     C2 C4
# R1 12 14
# R3 32 34
```

データフレーム要素参照

データフレームもシリーズと同様に、True と False からなるフィルター（ブーリアンベクトル）を使って要素を取得することができる。フィルターを使用する場合は、loc または iloc の両方を使用することができる。




```
import pandas as pd

d = pd.DataFrame([[1, 4],
                  [0, 1],
                  [1, 0],
                  [1, 3],
                  [0, 5]],
   index=['R1', 'R2', 'R3', 'R4', 'R5'],
 columns=['C1', 'C2'])


keep = (d.loc[:, 'C1'] > 0)

d.loc[keep, :]
#     C1 C2
# R1  1  4
# R3  1  0
# R4  1  3
```

データフレーム要素参照

参考

前出の .iloc および .loc 以外にも、.iat および .at を利用した要素参照や列名・行名属性を用いた参照方法などがある。

```
import pandas as pd

df = pd.DataFrame([[1, 4], [0, 1],
                   [1, 0], [1, 3],
                   [0, 5]],
  index=['R1','R2','R3','R4','R5'],
  columns=['C1','C2'])

df.iloc[2:4, 0]
df.loc[['R1', 'R3'], ['C1']]
df.iat[0, 1]
df.at['R2', 'C2']
df[0:1]
df[['C1', 'C2']]
df.C2
```

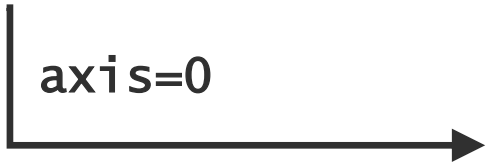
df.iloc	整数からなる位置番号を指定して、該当位置の要素を取得する。複数列や複数行の要素をまとめて取得できる。
df.loc	列名あるいは行名を指定して、該当位置の要素を取得する。複数列や複数行の要素をまとめて取得できる。
df.iat	.iloc と同じ使い方をする。ただし、1 つの要素しか取得できない。
df.at	.loc と同じ使い方をする。ただし、1 つの要素しか取得できない。
df[0:1]	行の位置番号をスライス表記で指定して、該当行の要素を取得する。複数行の要素をまとめて取得できる。
df[['C1']]	列の名前をリストで指定して、該当列の要素を取得する。複数列の要素をまとめて取得できる。ただし、スライス表記は使用できない。
df.C1	列の名前をオブジェクトの属性として、該当列の要素を取得することができる。

データフレーム / pd.concat

pd.concat 関数を使用することで、複数のデータフレームを結合させて 1 つのデータフレームにまとめることができる。pd.concat 関数の axis 引数を通して結合する次元方向を指定できる。

11	12	13	14
21	22	23	24

31	32	33	34
41	42	43	44



11	12	13	14
21	22	23	24
31	32	33	34
41	42	43	44

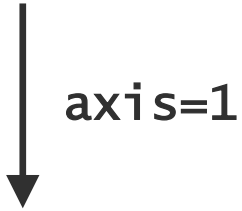
```
import pandas as pd

d1 = pd.DataFrame([[11, 12, 13, 14],
                   [21, 22, 23, 24]])
d2 = pd.DataFrame([[31, 32, 33, 34],
                   [41, 42, 43, 44]])

df = pd.concat([d1, d2])
df
#    0  1  2  3
# 0 11 12 13 14
# 1 21 22 23 24
# 2 31 32 33 34
# 3 41 42 43 44
```

データフレーム / pd.concat

pd.concat 関数を使用することで、複数のデータフレームを結合させて 1 つのデータフレームにまとめることができる。pd.concat 関数の axis 引数を通して結合する次元方向を指定できる。

11	12	13	14
21	22	23	24

31	32	33	34
41	42	43	44



11	12	13	14	31	32	33	34
21	22	23	24	41	42	43	44

```
import pandas as pd

d1 = pd.DataFrame([[11, 12, 13, 14],
                   [21, 22, 23, 24]])
d2 = pd.DataFrame([[31, 32, 33, 34],
                   [41, 42, 43, 44]])

df = pd.concat([d1, d2], axis=1)
df
#    0  1  2  3  0  1  2  3
# 0 11 12 13 14 31 32 33 34
# 1 21 22 23 24 41 42 43 44
```

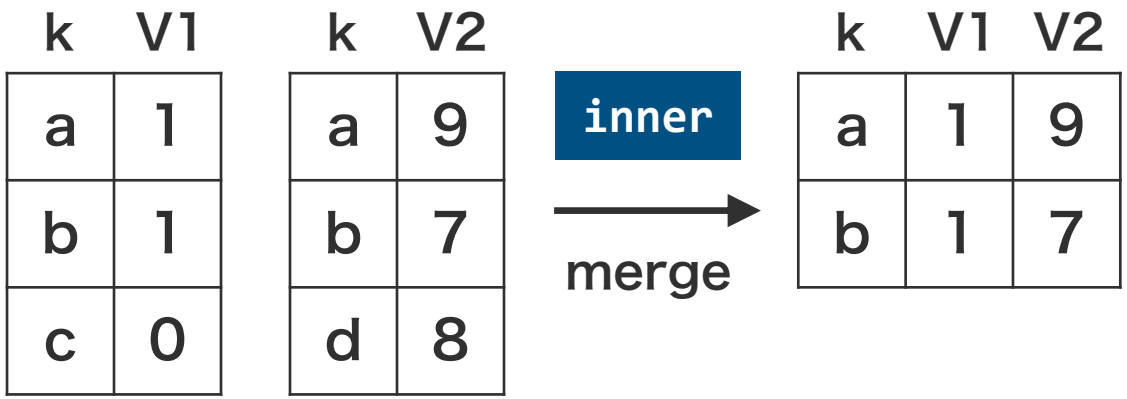
データフレーム / pd.concat

データフレームに index または列名が存在するとき、データフレーム同士が index と列名に基づいて結合される。結合後のデータフレームの行と列の並び順に十分に注意すること。

```
import pandas as pd

d1 = pd.DataFrame([[11, 12, 13, 14],
                   [21, 22, 23, 24]],
        index=['A', 'B'],
        columns=['a', 'b', 'c', 'd'])

d2 = pd.DataFrame([[31, 32, 33, 34],
                   [41, 42, 43, 44]],
        index=['X', 'Y'],
        columns=['a', 'b', 'c', 'e'])


df = pd.concat([d1, d2])
df
#     a  b  c    d    e
# A 11 12 13 14.0  NaN
# B 21 22 23 24.0  NaN
# X 31 32 33  NaN 34.0
# Y 41 42 43  NaN 44.0
```

データフレーム / pd.merge

複数のデータフレームを、特定の列の値に基づいて、マージするすことができる。このとき pd.merge 関数を使用する。



データフレームの結合を行うとき、キーとなる列に重複要素が含まれると、予期しない挙動になる場合があるため、十分に注意すること。

```
import pandas as pd

d1 = pd.DataFrame([['a', 1],
                   ['b', 1],
                   ['c', 0]],
                  columns=['k', 'V1'])

d2 = pd.DataFrame([['a', 9],
                   ['b', 7],
                   ['d', 8]],
                  columns=['k', 'V2'])


d = pd.merge(d1, d2)   # how='inner'
d
#   k V1 V2
# 0 a  1  9
# 1 b  1  7
```

データフレーム / merge

k	V1
a	1
b	1
c	0

k	V2
a	9
b	7
d	8

outer

merge

k	V1	V2
a	1	9
b	1	7
c	0	NaN
d	NaN	8

```
import pandas as pd

d1 = pd.DataFrame([['a', 1],
                   ['b', 1],
                   ['c', 0]],
                  columns=['k', 'V1'])

d2 = pd.DataFrame([['a', 9],
                   ['b', 7],
                   ['d', 8]],
                  columns=['k', 'V2'])


d = pd.merge(d1, d2, how='outer')
d
#    k   V1   V2
# 0  a  1.0  9.0
# 1  b  1.0  7.0
# 2  c  0.0  NaN
# 3  d  NaN  8.0
```

データフレーム / merge

k	V1
a	1
b	1
c	0

k	V2
a	9
b	7
d	8

left

merge

k	V1	V2
a	1	9
b	1	7
c	0	NaN

```
import pandas as pd

d1 = pd.DataFrame([['a', 1],
                   ['b', 1],
                   ['c', 0]],
                  columns=['k', 'V1'])

d2 = pd.DataFrame([['a', 9],
                   ['b', 7],
                   ['d', 8]],
                  columns=['k', 'V2'])


d = pd.merge(d1, d2, how='left')
d
#    k  V1   V2
# 0  a   1  9.0
# 1  b   1  7.0
# 2  c   0  NaN
```

データフレーム / merge

k	V1
a	1
b	1
c	0

k	V2
a	9
b	7
d	8

right

merge

k	V1	V2
a	1	9
b	1	7
d	NaN	8

```
import pandas as pd

d1 = pd.DataFrame([['a', 1],
                   ['b', 1],
                   ['c', 0]],
                  columns=['k', 'V1'])

d2 = pd.DataFrame([['a', 9],
                   ['b', 7],
                   ['d', 8]],
                  columns=['k', 'V2'])


d = pd.merge(d1, d2, how='right')
d
#    k   V1  V2
# 0  a  1.0   9
# 1  b  1.0   7
# 2  d  NaN   8
```

データフレーム / merge

マージ対象のデータフレームの基準列の列名が異なる場合は、pd.merge 関数の left_on および right_on 引数で指定する。

k	V1
a	1
b	1
c	0

f	V2
a	9
b	7
d	8

merge

k	V1	f	V2
a	1	a	9
b	1	b	7
c	0	NaN	NaN
NaN	NaN	d	8

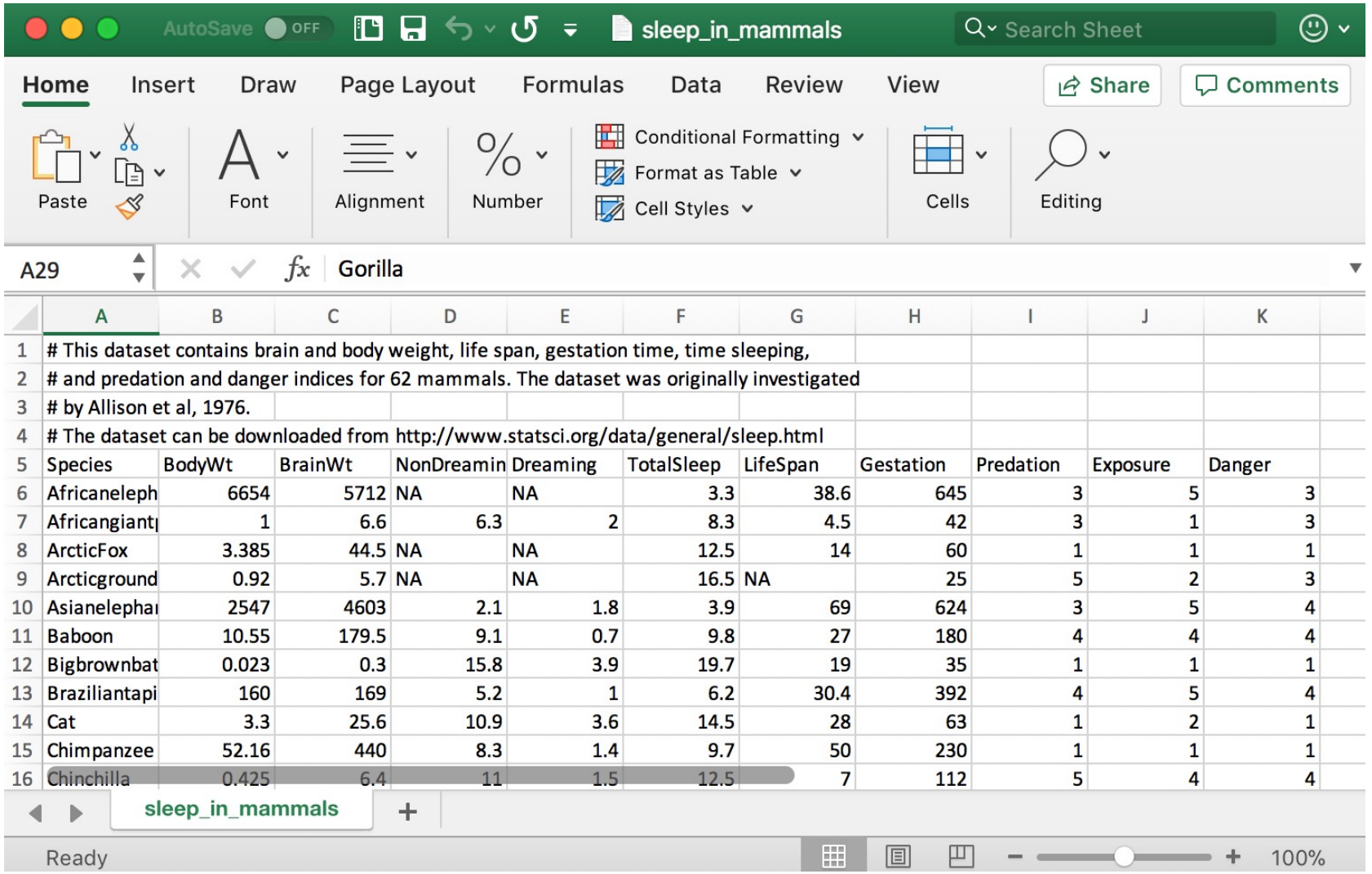
```
import pandas as pd

d1 = pd.DataFrame([['a', 1],
                   ['b', 1],
                   ['c', 0]],
                  columns=['k', 'V1'])

d2 = pd.DataFrame([['a', 9],
                   ['b', 7],
                   ['d', 8]],
                  columns=['f', 'V2'])

d = pd.merge(d1, d2, how='outer',
             left_on='k', right_on='f')
d
#      k   V1    f   V2
# 0    a  1.0    a  9.0
# 1    b  1.0    b  7.0
# 2    c  0.0  NaN  NaN
# 3  NaN  NaN    d  8.0
```

Pandas

- シリーズ
- データフレーム
- **表データ処理**

表データ

生物学で取り扱うデータは、一般的に、1 サンプル 1 行で記載されている。また、各行は複数の項目からなり、サンプルの属性が記載されいてる。このようなデータは、一般的にタブ区切りファイル（TSV）またはカンマ区切りファイル（CSV）のテキストファイルで保存される。



	A	B	C	D	E	F	G	H	I	J	K
1	# This dataset contains brain and body weight, life span, gestation time, time sleeping,										
2	# and predation and danger indices for 62 mammals. The dataset was originally investigated										
3	# by Allison et al, 1976.										
4	# The dataset can be downloaded from http://www.statsci.org/data/general/sleep.html										
5	Species	BodyWt	BrainWt	NonDreamin	Dreaming	TotalSleep	LifeSpan	Gestation	Predation	Exposure	Danger
6	Africaneleph	6654	5712	NA	NA	3.3	38.6	645	3	5	3
7	Africangiant	1	6.6	6.3	2	8.3	4.5	42	3	1	3
8	ArcticFox	3.385	44.5	NA	NA	12.5	14	60	1	1	1
9	Arcticground	0.92	5.7	NA	NA	16.5	NA	25	5	2	3
10	Asianelephar	2547	4603	2.1	1.8	3.9	69	624	3	5	4
11	Baboon	10.55	179.5	9.1	0.7	9.8	27	180	4	4	4
12	Bigbrownbat	0.023	0.3	15.8	3.9	19.7	19	35	1	1	1
13	Braziliantapi	160	169	5.2	1	6.2	30.4	392	4	5	4
14	Cat	3.3	25.6	10.9	3.6	14.5	28	63	1	2	1
15	Chimpanzee	52.16	440	8.3	1.4	9.7	50	230	1	1	1
16	Chinchilla	0.425	6.4	11	1.5	12.5	7	112	5	4	4

表データ

コメント

データ

```
# This dataset contains brain and body weight, life span, gestation time,
# and predation and danger indices for 62 mammals. The dataset was origin
# by Allison et al, 1976.
# The dataset can be downloaded from http://www.statsci.org/data/general/
Species BodyWt  BrainWt NonDreaming     Dreaming         TotalSleep       L
Africanelephant 6654    5712    NA      NA      3.3      38.6    645      3
Africangiantpouchedrat  1       6.6     6.3     2        8.3     4.5      4
ArcticFox       3.385   44.5    NA      NA      12.5     14      60       1
Arcticgroundsquirrel    0.92    5.7     NA      NA       16.5    NA       2
Asianelephant   2547    4603    2.1     1.8     3.9      69      624      3
Baboon  10.55   179.5   9.1     0.7     9.8      27      180     4        4
Bigbrownbat     0.023   0.3     15.8    3.9     19.7     19      35       1
Braziliantapir  160     169     5.2     1       6.2      30.4    392      4
Cat     3.3     25.6    10.9    3.6     14.5     28      63      1        2
Chimpanzee      52.16   440     8.3     1.4     9.7      50      230      1
Chinchilla      0.425   6.4     11      1.5     12.5     7       112      5
Cow     465     423     3.2     0.7     3.9      30      281     5        5
```

https://aabbdd.jp/notes/data/sleep_in_mammals.txt

表データ

属性（特徴量）

ヘッダー →

サンプル

Species	BodyWt	BrainWt	NonDreaming		Dreaming		TotalSleep		L
Africanelephant	6654	5712	NA		NA	3.3	38.6	645	3
Africangiantpouchedrat	1	欠損値	6.6		6.3	2	8.3	4.5	4
ArcticFox	3.385	44.5	NA		NA	12.5	14	60	1
Arcticgroundsquirrel		0.92	5.7		NA	NA	16.5	NA	2
Asianelephant	2547	4603	2.1		1.8	3.9	69	624	3
Baboon	10.55	179.5	9.1		0.7	9.8	27	180	4
Bigbrownbat		0.023	0.3		15.8	3.9	19.7	19	1
Braziliantapir		160	169		5.2	1	6.2	30.4	392
Cat	3.3	25.6	10.9		3.6	14.5	28	63	1
Chimpanzee		52.16	440		8.3	1.4	9.7	50	230
Chinchilla		0.425	6.4		11	1.5	12.5	7	112
Cow	465	423	3.2		0.7	3.9	30	281	5

https://aabbdd.jp/notes/data/sleep_in_mammals.txt

表データ読み込み

Pandas の read_table 関数を使うことで、CSV または TSV データを読み込むことができる。

```
import pandas as pd

f = 'sleep_in_mammals.txt'

d = pd.read_csv(f)
```

```
 # This dataset contains brain and body weight   life span   gestation time   time sleeping  Unnamed: 4
0   # and predation and danger indices for 62 mamm...         NaN              NaN             NaN         NaN
1                                 # by Allison et al      1976.0              NaN             NaN         NaN
2   # The dataset can be downloaded from http://ww...         NaN              NaN             NaN         NaN
3   Species\tBodyWt\tBrainWt\tNonDreaming\tDreamin...         NaN              NaN             NaN         NaN
4   Africanelephant\t6654\t5712\tNA\tNA\t3.3\t38.6...         NaN              NaN             NaN         NaN
..                                                ...         ...              ...             ...         ...
61  Treehyrax\t2\t12.3\t4.9\t0.5\t5.4\t7.5\t200\t3...         NaN              NaN             NaN         NaN
62  Treeshrew\t0.104\t2.5\t13.2\t2.6\t15.8\t2.3\t4...         NaN              NaN             NaN         NaN
63  Vervet\t4.19\t58\t9.7\t0.6\t10.3\t24\t210\t4\t...         NaN              NaN             NaN         NaN
64  Wateropossum\t3.5\t3.9\t12.8\t6.6\t19.4\t3\t14...         NaN              NaN             NaN         NaN
65  Yellow-belliedmarmot\t4.05\t17\tNA\tNA\tNA\t13...         NaN              NaN             NaN         NaN
```

read_csv 関数のオプションを正しく指定しないと、古いバージョンの Pandas ではエラーが起き、新しいバージョンの Pandas ではエラーは起きないがデータを正しく認識できない。

https://aabbdd.jp/notes/data/sleep_in_mammals.txt

表データ読み込み

Pandas の read_table 関数を使うことで、CSV または TSV データを読み込むことができる。データを正しく読み込むには、read_table 関数に、コメント行を明示し、ヘッダー行の有無、区切り文字の種類を正しく指定する必要がある。

```
import pandas as pd

f = 'sleep_in_mammals.txt'

d = pd.read_csv(f,
                comment='#',
                header=0,
                sep='\t')
```

バックスラッシュは特別な意味を持つ文字である。バックスラッシュの後に続く1文字は、特別な意味を持つ。例えば 't' は英文字の t を表すが、'\t' はタブを表す。

https://aabbdd.jp/notes/data/sleep_in_mammals.txt

表データ読み込み

```
import pandas as pd

f = 'sleep_in_mammals.txt'

d = pd.read_csv(f, comment='#', header=0, sep='\t')

d.shape
# (62, 11)

d.head()
#                    Species    BodyWt  BrainWt  ...  Predation  Exposure  Danger
# 0         Africanelephant  6654.000   5712.0  ...          3         5       3
# 1  Africangiantpouchedrat     1.000      6.6  ...          3         1       3
# 2                ArcticFox     3.385     44.5  ...          1         1       1
# 3    Arcticgroundsquirrel     0.920      5.7  ...          5         2       3
# 4            Asianelephant  2547.000   4603.0  ...          3         5       4
# [5 rows x 11 columns]
```

https://aabbdd.jp/notes/data/sleep_in_mammals.txt

表データ読み込み

```
import pandas as pd

f = 'sleep_in_mammals.txt'

d = pd.read_csv(f, comment='#', header=0, sep='\t', index_col=0)

d.shape
# (62, 10)

d.head()
#                          BodyWt  BrainWt  ...  Exposure  Danger
# Species                                   ...
# Africanelephant        6654.000   5712.0  ...         5       3
# Africangiantpouchedrat    1.000      6.6  ...         1       3
# ArcticFox                 3.385     44.5  ...         1       1
# Arcticgroundsquirrel      0.920      5.7  ...         2       3
# Asianelephant          2547.000   4603.0  ...         5       4
# [5 rows x 10 columns]
```

特定の列をデータの一部ではなくて、行名として取り組むこともできる。

https://aabbdd.jp/notes/data/sleep_in_mammals.txt

表データ / 行名と列名

```
import pandas as pd

f = 'sleep_in_mammals.txt'

d = pd.read_csv(f, comment='#', header=0, sep='\t', index_col=0)

d.index
# Index(['Africanelephant', 'Africangiantpouchedrat', 'ArcticFox',
#        'Arcticgroundsquirrel', 'Asianelephant', 'Baboon', 'Bigbrownbat',
#        'Braziliantapir', 'Cat', 'Chimpanzee', 'Chinchilla', 'Cow',
#        ...
#        'Treeshrew', 'Vervet', 'Wateropossum', 'Yellow-belliedmarmot'],
#       dtype='object', name='Species')

d.columns
# Index(['BodyWt', 'BrainWt', 'NonDreaming', 'Dreaming', 'TotalSleep',
#        'LifeSpan', 'Gestation', 'Predation', 'Exposure', 'Danger'],
#       dtype='object')
```

https://aabbdd.jp/notes/data/sleep_in_mammals.txt

表データ / 要素の取得

```
import pandas as pd

f = 'sleep_in_mammals.txt'

d = pd.read_csv(f, comment='#', header=0, sep='\t', index_col=0)

d.iloc[0:3, 0:5]
#                          BodyWt  BrainWt  NonDreaming  Dreaming  TotalSleep
# Species
# Africanelephant         6654.000   5712.0          NaN       NaN         3.3
# Africangiantpouchedrat     1.000      6.6          6.3       2.0         8.3
# ArcticFox                  3.385     44.5          NaN       NaN        12.5

d.iloc[0:2, :]
#                          BodyWt  BrainWt  NonDreaming  ... Exposure  Danger
# Species                                                ...
# Africanelephant         6654.000   5712.0          NaN  ...        5       3
# Africangiantpouchedrat     1.000      6.6          6.3  ...        1       3
```

https://aabbdd.jp/notes/data/sleep_in_mammals.txt

表データ / 要素の取得

```
import pandas as pd

f = 'sleep_in_mammals.txt'

d = pd.read_csv(f, comment='#', header=0, sep='\t', index_col=0)

species = ['Cat', 'Rat', 'Cow', 'Pig']
features = ['BodyWt', 'BrainWt', 'TotalSleep', 'LifeSpan']

d.loc[species, features]
#          BodyWt  BrainWt  TotalSleep  LifeSpan
# Species
# Cat        3.30     25.6        14.5      28.0
# Rat        0.28      1.9        13.2       4.7
# Cow      465.00    423.0         3.9      30.0
# Pig      192.00    180.0         8.4      27.0
```

https://aabbdd.jp/notes/data/sleep_in_mammals.txt

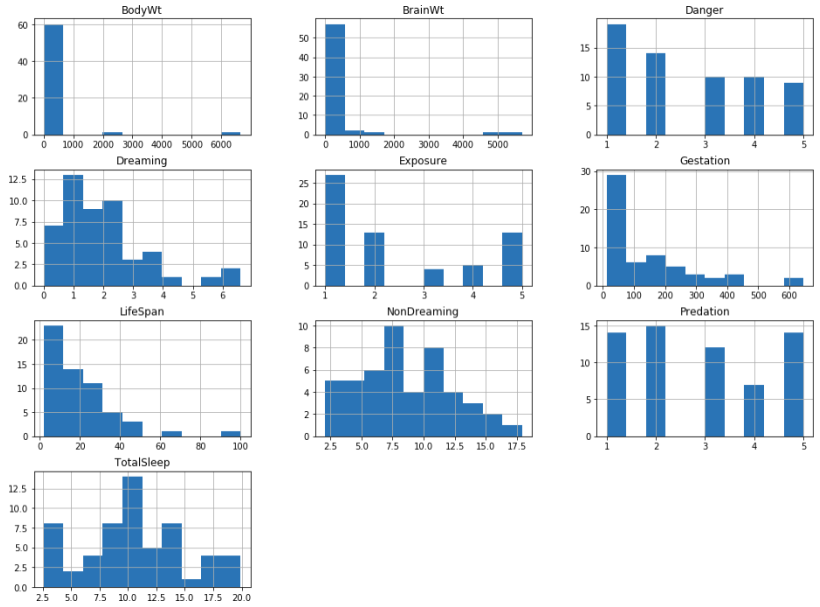
表データ読み込み / データ要約

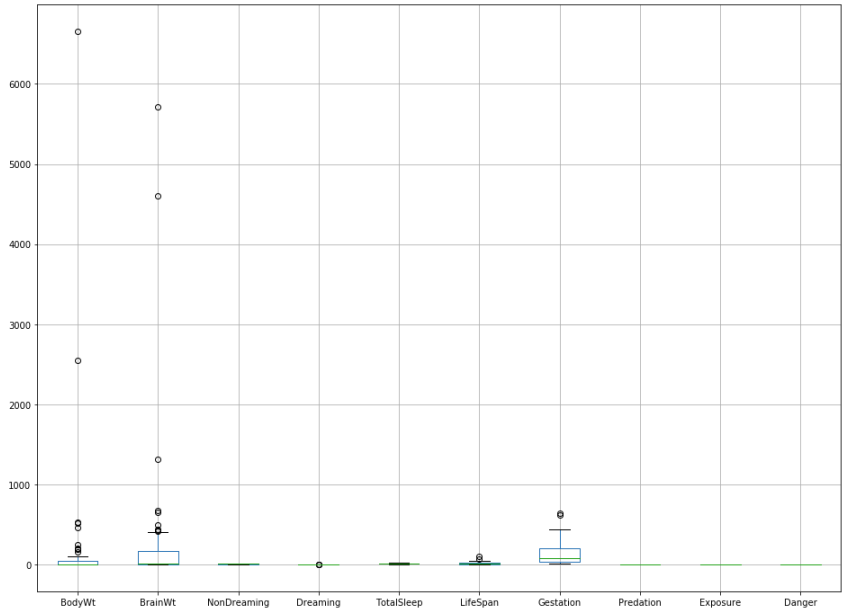
```
import pandas as pd

f = 'sleep_in_mammals.txt'

d = pd.read_csv(f, comment='#', header=0, sep='\t', index_col=0)

d.describe()
#              BodyWt      BrainWt  NonDreaming  ...  Predation   Exposure      Danger
# count    62.000000    62.000000    48.000000  ...  62.000000  62.000000  62.000000
# mean    198.789984   283.134194     8.672917  ...   2.870968   2.419355   2.612903
# std     899.158011   930.278942     3.666452  ...   1.476414   1.604792   1.441252
# min       0.005000     0.140000     2.100000  ...   1.000000   1.000000   1.000000
# 25%       0.600000     4.250000     6.250000  ...   2.000000   1.000000   1.000000
# 50%       3.342500    17.250000     8.350000  ...   3.000000   2.000000   2.000000
# 75%      48.202500   166.000000    11.000000  ...   4.000000   4.000000   4.000000
# max    6654.000000  5712.000000    17.900000  ...   5.000000   5.000000   5.000000
# [8 rows x 10 columns]
```

https://aabbdd.jp/notes/data/sleep_in_mammals.txt

表データ読み込み / データ可視化

```
import pandas as pd

f = 'sleep_in_mammals.txt'

d = pd.read_csv(f, comment='#', header=0, sep='\t', index_col=0)

d.hist()
d.boxplot()
```





※ 一部の属性の値の範囲が大きいので、対数化してからグラフに描くと全体の傾向を掴めやすくなる。

※ matplotlib でグラフを描いた方が一般的であるので、ここでは Pandas の視覚化機能を取り上げない。

https://aabbdd.jp/notes/data/sleep_in_mammals.txt

問題 P2-1

diversity_galapagos.txt を Pandas で読み込み、面積（Area）が最も大きい島の名前とその面積を答えよ。

```
import pandas as pd

f = 'diversity_galapagos.txt'
d = pd.read_csv(f, comment='#', header=0, sep='\t', index_col=0)
```

https://aabbdd.jp/notes/data/diversity_galapagos.txt

ヒント

シリーズ s の最大値は max(s) で求めることができる。

問題 P2-2

diversity_galapagos.txt を Pandas で読み込み、 各島における面積あたりの種数を求めよ。

```
import pandas as pd

f = 'diversity_galapagos.txt'
d = pd.read_csv(f, comment='#', header=0, sep='\t', index_col=0)
```

https://aabbdd.jp/notes/data/diversity_galapagos.txt

ファイル書き出し

Pandas では、データを CSV または TSV ファイルに書き出すとき、to_csv 関数（メソッド）を使う。この際に、区切り文字、行名の有無、列名の有無を指定することができる。また、欠損値を特定の文字に変換することもできる。

```
import pandas as pd
f = 'sleep_in_mammals.txt'

d = pd.read_csv(f, comment='#', header=0, sep='\t',
                index_col=0)

d.to_csv('o.txt', sep=',', header=False, index=False)
```

```
6654.0,5712.0,,,3.3,38.6,645.0,3,5,3
1.0,6.6,6.3,2.0,8.3,4.5,42.0,3,1,3
3.385,44.5,,,12.5,14.0,60.0,1,1,1
0.92,5.7,,,16.5,,25.0,5,2,3
2547.0,4603.0,2.1,1.8,3.9,69.0,624.0,3,5,4
10.55,179.5,9.1,0.7,9.8,27.0,180.0,4,4,4
0.023,0.3,15.8,3.9,19.7,19.0,35.0,1,1,1
160.0,169.0,5.2,1.0,6.2,30.4,392.0,4,5,4
3.3,25.6,10.9,3.6,14.5,28.0,63.0,1,2,1
52.16,440.0,8.3,1.4,9.7,50.0,230.0,1,1,1
```

ファイル書き出し

Pandas では、データを CSV または TSV ファイルに書き出すとき、to_csv 関数（メソッド）を使う。この際に、区切り文字、行名の有無、列名の有無を指定することができる。また、欠損値を特定の文字に変換することもできる。

```
import pandas as pd
f = 'sleep_in_mammals.txt'

d = pd.read_csv(f, comment='#', header=0, sep='\t',
                   index_col=0)

d.to_csv('o.txt', sep=',', header=True, index=True)
```

```
Species,BodyWt,BrainWt,NonDreaming,Dreaming,TotalSleep,LifeSpan,Gestation,Predation,Exposure
Africanelephant,6654.0,5712.0,,,3.3,38.6,645.0,3,5,3
Africangiantpouchedrat,1.0,6.6,6.3,2.0,8.3,4.5,42.0,3,1,3
ArcticFox,3.385,44.5,,,12.5,14.0,60.0,1,1,1
Arcticgroundsquirrel,0.92,5.7,,,16.5,,25.0,5,2,3
Asianelephant,2547.0,4603.0,2.1,1.8,3.9,69.0,624.0,3,5,4
Baboon,10.55,179.5,9.1,0.7,9.8,27.0,180.0,4,4,4
Bigbrownbat,0.023,0.3,15.8,3.9,19.7,19.0,35.0,1,1,1
Braziliantapir,160.0,169.0,5.2,1.0,6.2,30.4,392.0,4,5,4
Cat,3.3,25.6,10.9,3.6,14.5,28.0,63.0,1,2,1
```

ファイル書き出し

Pandas では、データを CSV または TSV ファイルに書き出すとき、to_csv 関数（メソッド）を使う。この際に、区切り文字、行名の有無、列名の有無を指定することができる。また、欠損値を特定の文字に変換することもできる。

```
import pandas as pd
f = 'sleep_in_mammals.txt'

d = pd.read_csv(f, comment='#', header=0, sep='\t',
                index_col=0)

d.to_csv('o.txt', sep='\t', header=True, index=True)
```

Species	BodyWt	BrainWt	NonDreaming		Dreaming		TotalSleep		LifeSpan		Gestati
Africanelephant	6654.0	5712.0			3.3		38.6	645.0	3	5	3
Africangiantpouchedrat	1.0	6.6	6.3		2.0		8.3	4.5	42.0	3	1
ArcticFox	3.385	44.5			12.5		14.0	60.0	1	1	1
Arcticgroundsquirrel		0.92	5.7				16.5		25.0	5	2
Asianelephant	2547.0	4603.0	2.1	1.8	3.9		69.0	624.0	3	5	4
Baboon	10.55	179.5	9.1	0.7	9.8		27.0	180.0	4	4	4
Bigbrownbat		0.023	0.3	15.8	3.9		19.7	19.0	35.0	1	1
Braziliantapir	160.0	169.0	5.2	1.0	6.2		30.4	392.0	4	5	4
Cat	3.3	25.6	10.9	3.6	14.5		28.0	63.0	1	2	1

ファイル書き出し

Pandas では、データを CSV または TSV ファイルに書き出すとき、to_csv 関数（メソッド）を使う。この際に、区切り文字、行名の有無、列名の有無を指定することができる。また、欠損値を特定の文字に変換することもできる。

```
import pandas as pd
f = 'sleep_in_mammals.txt'

d = pd.read_csv(f, comment='#', header=0, sep='\t',
                index_col=0)

d.to_csv('o.txt', sep='\t', header=True, index=True)
         na_rep= 'NA')
```

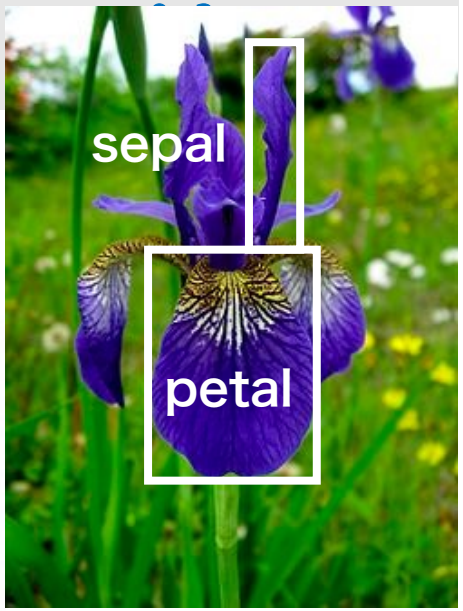
```
Species BodyWt  BrainWt NonDreaming     Dreaming        TotalSleep      LifeSpan        Gestati
Africanelephant 6654.0  5712.0  NA      NA      3.3     38.6    645.0   3       5       3
Africangiantpouchedrat  1.0     6.6     6.3     2.0     8.3     4.5     42.0    3       1
ArcticFox       3.385   44.5    NA      NA      12.5    14.0    60.0    1       1       1
Arcticgroundsquirrel    0.92    5.7     NA      NA      16.5    NA      25.0    5       2
Asianelephant   2547.0  4603.0  2.1     1.8     3.9     69.0    624.0   3       5       4
Baboon  10.55   179.5   9.1     0.7     9.8     27.0    180.0   4       4       4
Bigbrownbat     0.023   0.3     15.8    3.9     19.7    19.0    35.0    1       1       1
Braziliantapir  160.0   169.0   5.2     1.0     6.2     30.4    392.0   4       5       4
Cat     3.3     25.6    10.9    3.6     14.5    28.0    63.0    1       2       1
```

空白がNAに置換される

グループ演算

Pandas のデータフレームでは、ある列の値に基づいて、全データをいくつかのサブセット分け、それぞれのサブセットに対して平均や分散などを計算する機能が提供されている。

```
import pandas as pd
f = 'iris.txt'
d = pd.read_csv(f, header=0, sep='\t')

d.head()
#    ID  Sepal.Length  Sepal.Width  Petal.Length  Petal.Width Species
# 0   1           5.1          3.5           1.4          0.2  setosa
# 1   2           4.9          3.0           1.4          0.2  setosa
# 2   3           4.7          3.2           1.3          0.2  setosa
# 3   4           4.6          3.1           1.5
# 4   5           5.0          3.6           1.4
```

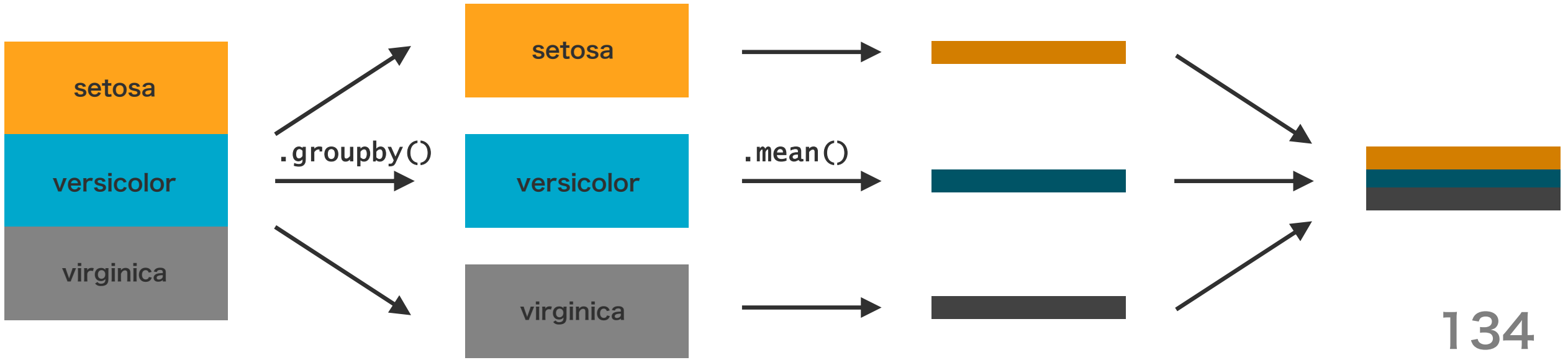


iris データセットは、3種のアヤメ(setosa・versicolor・virginica)に対して、萼 sepal の長さ length と幅 width、花弁 petal の長さ length と幅 width を測定したデータである。各種には 50 個体のデータ含まれ、3種で計150個体のデータが含まれている。

https://aabbdd.jp/notes/data/iris.txt

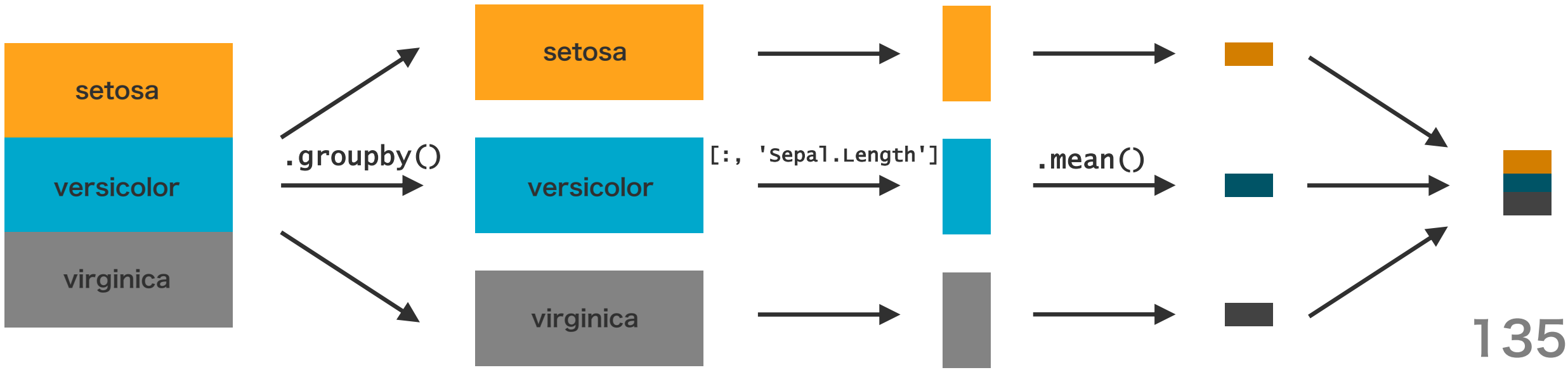
グループ演算 / groupby

```
import pandas as pd
f = 'iris.txt'
d = pd.read_csv(f, header=0, sep='\t')

d.groupby('Species').mean()
#                ID  Sepal.Length  Sepal.Width  Petal.Length  Petal.Width
# Species
# setosa       25.5         5.006        3.428         1.462        0.246
# versicolor   75.5         5.936        2.770         4.260        1.326
# virginica   125.5         6.588        2.974         5.552        2.026
```



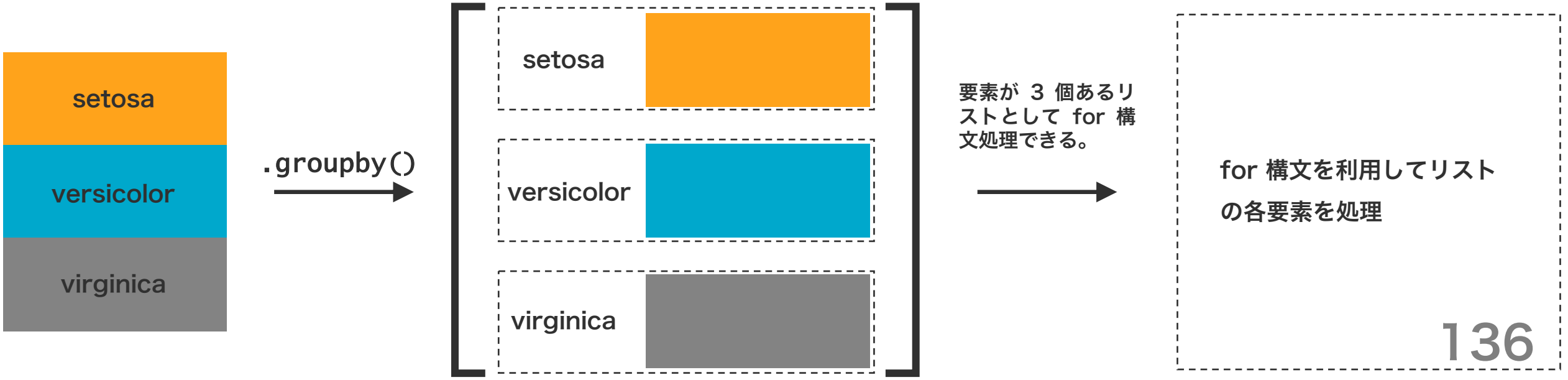
グループ演算 / groupby

```
import pandas as pd
f = 'iris.txt'
d = pd.read_csv(f, header=0, sep='\t')

d.groupby('Species').loc[:, 'Sepal.Length'].mean()
# Species
# setosa         5.006
# versicolor     5.936
# virginica      6.588
# Name: Sepal.Length, dtype: float64
```



グループ演算 / groupby

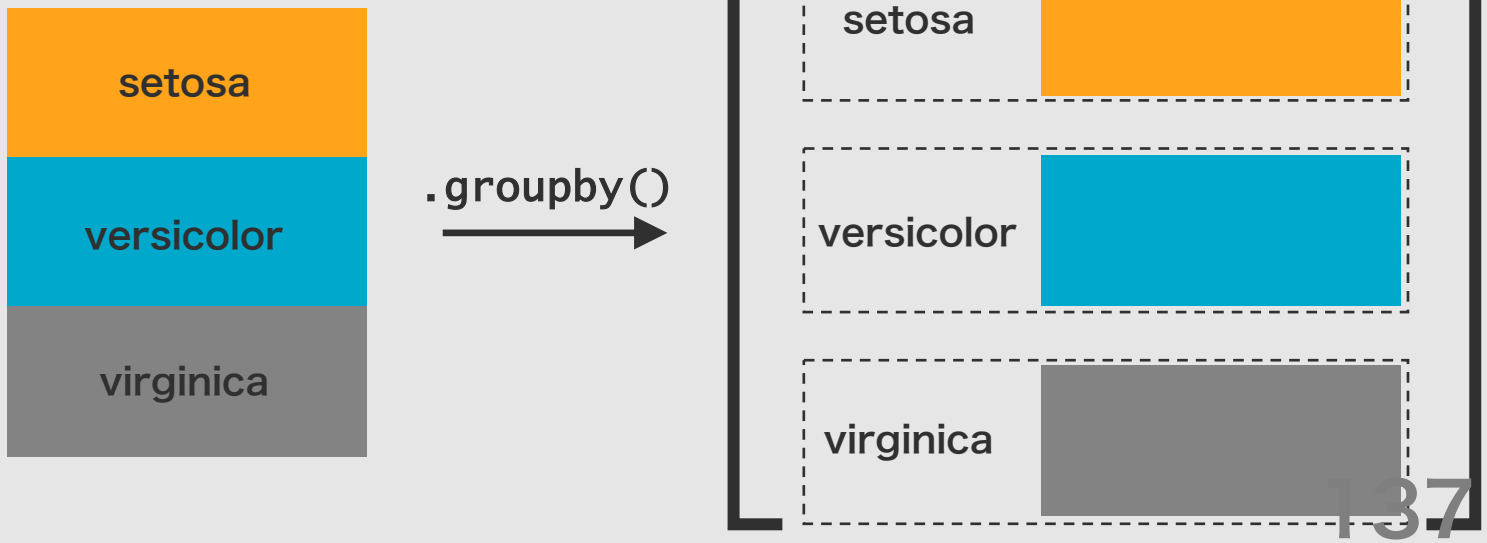
```
import pandas as pd
f = 'iris.txt'
d = pd.read_csv(f, header=0, sep='\t')

for gname, subset in d.groupby('Species'):
    print(gname)
    print(subset.head())
```



グループ演算 / groupby

```
import pandas as pd
f = 'iris.txt'
d = pd.read_csv(f, header=0, sep='\t')

for gname, subset in d.groupby('Species'):
    M = subset.iloc[:, 1:5].max(axis=0)
    m = subset.iloc[:, 1:5].min(axis=0)
    print(gname)
    print(M - m)

# setosa
# Sepal.Length 1.5
# Sepal.Width 2.1
# Petal.Length 0.9
# Petal.Width 0.5
# versicolor
# Sepal.Length 2.1
# Sepal.Width 1.4
# ...
# ...
```

2 列目から 6 列目の部分列を抽出し、各列の最大値を計算する。



グループ演算 / apply

```
import pandas as pd
f = 'iris.txt'
d = pd.read_csv(f, header=0, sep='\t')


def calc_diff(df):
    M = df.iloc[:, 1:5].max(axis=0)
    m = df.iloc[:, 1:5].min(axis=0)
    return M - m


d.groupby('Species').apply(calc_diff)
#             Sepal.Length  Sepal.Width  Petal.Length  Petal.Width
# Species
# setosa               1.5          2.1           0.9          0.5
# versicolor           2.1          1.4           2.1          0.8
# virginica            3.0          1.6           2.4          1.1
```

for 構文を利用したグループ演算を関数化することで、高速な apply 関数を利用できるようにある。

問題 P3-1

iris.txtを読み込み、各種におけるにおける、花弁 petal の長さ length と幅 width の比率の平均値を求めよ。

```
import pandas as pd

f = 'iris.txt'
d = pd.read_csv(f, header=0, sep='\t')
```

https://aabbdd.jp/notes/data/iris.txt