# 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、農学生命科学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

OCT
03 13:15–16:30

## Python 基礎

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。

OCT
10 13:15–16:30

## テキストデータ処理

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。

OCT
17 13:15–16:30

## データ分析

第 3 回目の授業では、Python ライブラリー（NumPy や Pandas）を利用して、CSV ファイルの処理などのデータ分析やデータ可視化を中心に取り上げる。

OCT
24 13:15–16:30

## スマート農業

Python のライブラリー（PyTorch 等）を利用して、深層学習による物体分類や物体検出モデルを実装する例を示す。

# 農学生命情報科学特論Ⅰ

② 

- テキスト処理
- ファイル処理
- NumPy

# 農学生命情報科学特論Ⅰ

② テキスト処理

○ ファイル処理

○ NumPy

# テキストデータ

テキストデータはアルファベットや仮名などの文字からなるデータを指す。Python のオブジェクトには、数値の他にアルファベット・漢字・仮名などの文字を代入することもできる。文字をオブジェクトに代入するとき、その文字が、オブジェクトの名前ではなく、文字のデータであることを明示するために、文字データの両側を引用符で囲む。

```
s = 'a'


t = 'B'


u = 't'
u
# 't'


v = t
v
# 'B'
```

t が引用符で囲まれているため、文字データとしての t がオブジェクト u に代入される。

t が引用符で囲まれていないため、t はオブジェクトである。オブジェクト t の内容がオブジェクト v に代入される。

# テキストデータ

単語や文章などのような文字列も、1 つのオブジェクトに代入できる。この場合、1 つのオブジェクトに、複数の文字データが保存されている、と捉えることができるため、このオブジェクトをリストのように扱うことができる。添字を指定して、特定の位置の文字を取り出したり、スライスして部分文字列を取り出したりすることができる。また、for 構文を使用して、文字列中の文字を 1 つずつ順に取り出すこともできる。

```
s = 'Smile. Tomorrow will be worse.'

s[0]
# 'S'



s[7:15]
# 'Tommorow'



for t in s:
    print(t)
# 'S'
# 'm'
# 'i'
# 'l'
# 'e'
# ......
```

# テキストデータ

文字列を保持しているオブジェクトに対して、足算が定義されている。そのため、+ 演算子を使用することで、複数の文字列を連結することができる。

```
s1 = 'Smile.'
s2 = 'Tomorrow will be worse.'
s3 = ' '

s = s1 + s2
s
# 'Smile.Tomorrow will be worse.'


s = s1 + s3 + s2
s
# 'Smile. Tomorrow will be worse.'


s = s1 + ' ' + s2
s
# 'Smile. Tomorrow will be worse.'
```

# 問題 T1-1

赤色で書かれたオブジェクトが保持している値を答えよ。

```
s1 = 'anything that can go wrong'
s2 = 'it will go wrong'



s1[:8]



s2[3:]



s3 = s1 + ',' + s2[3:]
s3
```

```
s1 = 'left to themselves'
s2 = 'things tend to go'
s3 = 'from bad to better'



s = s1 + ', ' + s2
s = s + ' ' + s3[:12] + 'worse.'
s
```

# 問題 T1-2

与えられた塩基配列の中から ATG を検索し、ATG が見つかればその位置を、そうでなければ -1 を出力するプログラムを for/while 構文や if 構文で作成せよ。

```
s = 'CCACAGTCATGTGTCAGTCGTAAGT'



# 8
```

```
s = 'CATTGTGTCACAGCCAGTCGTAAGT'



# -1
```

# 問題 T1-3

与えられた塩基配列中の A、C、G、T の出現回数および出現確率を求めよ。

```
s = 'AAAGGTCTTT'




# A: 3, G: 2, C: 1, T: 4
```

与えられた塩基配列に含まれる AA、AC、・・・、TT のような 2 文字パターンの出現回数を求めよ。

```
s = 'AAAGGTCTTT'
     ==
      ==
       ==
        ==



# AA: 2, AG: 1, GG: 1, GT: 1,
# TC: 1, CT: 1, TT: 2
```

# 問題 T1-4

for/while 構文や if 構文を使用して、与えられた塩基配列の相補鎖を求めよ。

```
s = 'AAAGGTC'
c = ''









c
# 'TTTCCAG'
```

for/while 構文や if 構文を使用して、与えられた塩基配列の逆相補鎖を求めよ。

```
s = 'AAAGGTC'
r = ''









r
# 'GACCTTT'
```

# 文字列操作関数

文字列の操作には、連結、分割、検索、置換などがある。これらの操作は、次の関数（文字列メソッド）で行える。

| | |
|---|---|
| `a + b` | 文字列 a の後ろに文字列 b を連結する。 |
| `a.split(',')` | コンマ , を区切り文字として、文字列を分割して、部分文字列のリストに変換する。 |
| `a.find('ATG')` | 文字列 a の先頭から ATG を探す。見つかればその位置の添字を返し、そうでなければ -1 を返す。 |
| `a.replace('TAG', '*')` | 文字列 a 中ににある部分文字列 TAG を * に置き換える。 |

```
s = '21,31,41,51,61'


s.split(',')
# ['21', '31', '41', '51', '61']



s.find('41')
# 6


s.find('71')
# -1


s.replace(',', ';')
# '21;31;41;51;61'
```

# 文字と数値の交互変換

Python の世界で、文字と数値の扱い方が異なる。文字と文字の足し算では文字同士が連結され、数値と数値の足し算では数学的に和が計算される。文字と数値の足し算は定義されておらず、エラーが出る。

```
a = '12'
b = 21


a + b
# TypeError: can only concatenate str
(not "int") to str
```

文字列 a に、整数 b を連結できないことを示すエラー。

```
b + a
# TypeError: unsupported operand
type(s) for +: 'int' and 'str'
```

整数 b に文字列 a を足せないことを示すエラー。

# 文字から数値への変換

文字列を整数または小数に変換することができる。整数への変換は int 関数を利用し、小数への変換は float 関数を利用する。これらの関数は、すべての文字列を整数・小数に変換できるわけではない。変換できない場合は、エラーが起こる。

```
a1 = '12'
a2 = '12.345'
a3 = '1.23e6'

int(a1)
# 12

int(a2)
# ValueError: invalid literal for int()
with base 10: '12.345'

float(a1)
# 12.0

float(a2)
# 12.345

float(a3)
# 1230000.0
```

# 数値から文字への変換

数値から文字への変換は str 関数を使用する。基本的に、すべての数値を文字に変換できる。桁数の多い小数を文字に変換するとき、桁落ちが発生する場合がある。

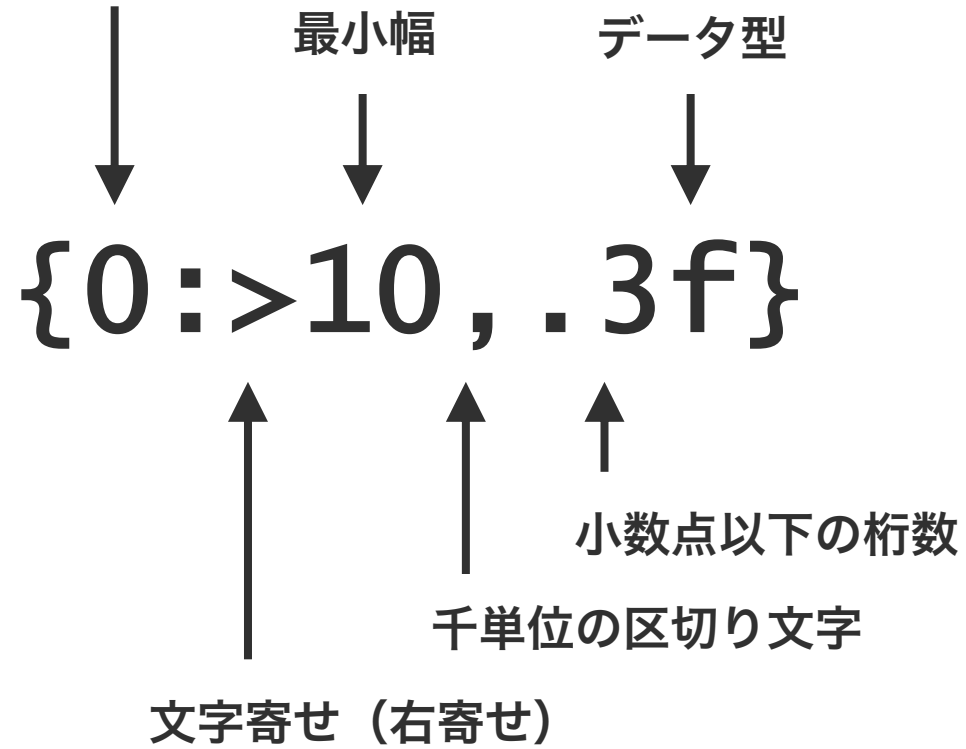
```
b1 = 12
b2 = 12.345
b3 = 12.34567890123456789 01
b4 = 1.23e6


str(b1)
# '12'


str(b2)
# '12.345'


str(b3)
# '12.345678901234567'


str(b4)
# '1230000.0'
```

# 数値から文字への変換

参考

数値から文字への変換は、str 関数のほかに format 関数も用意されている。format 関数を使うことで、有効桁数を指定したり、ゼロ埋めしたりすることができる。

埋め込みインデックス



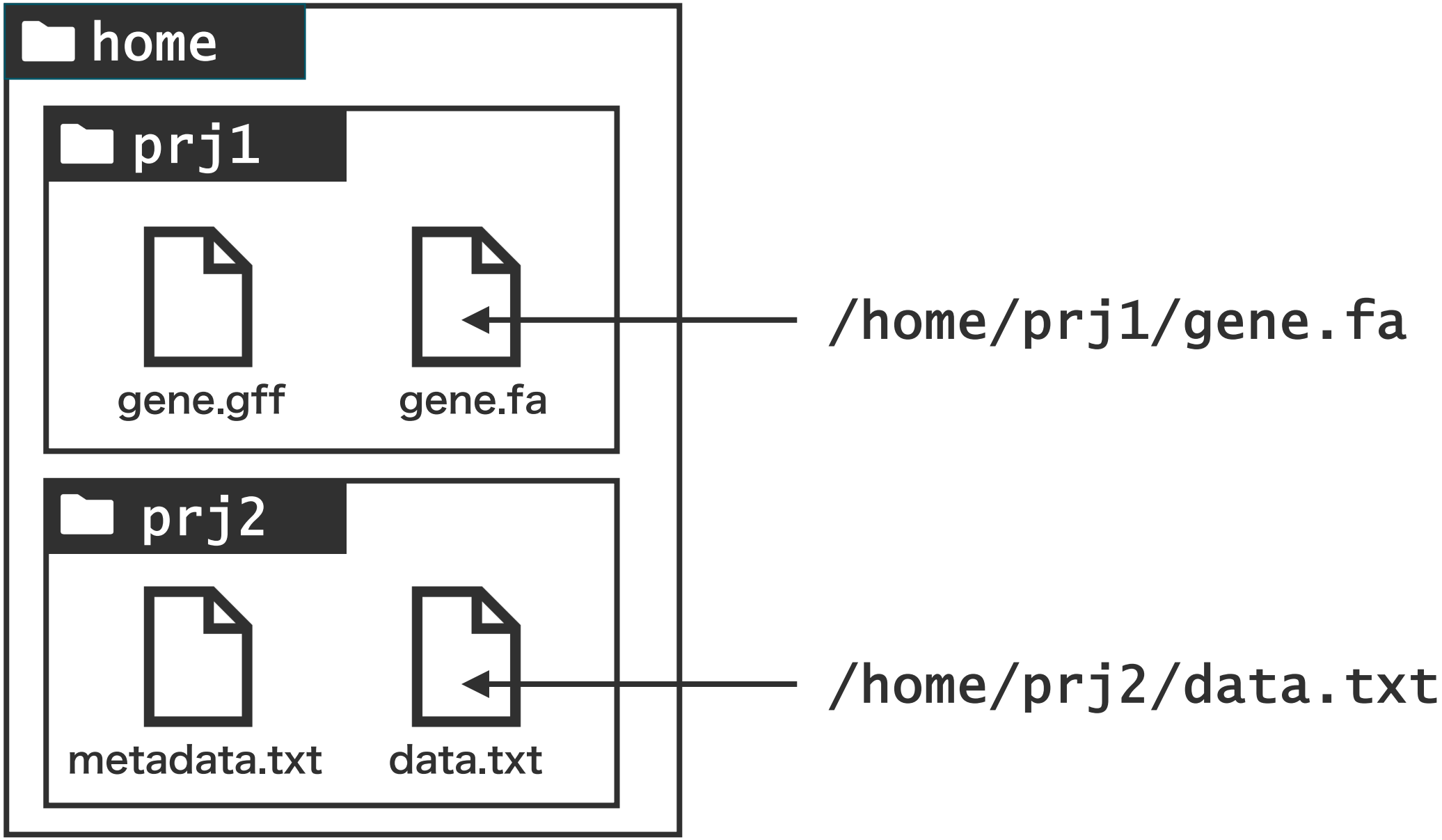

```
b1 = 12
b2 = 12.3456789


'{0:4d}'.format(b1)
# '  12'

'{0:04d}'.format(b1)
# '0012'

'{0:.3f}'.format(b1)
# '12.000'

'{0:.3f}'.format(b2)
# '12.345'

'{0:.2e}'.format(b2)
# '1.23e+01'
```

# 農学生命情報科学特論Ⅰ

② 

- ○ テキスト処理
- ○ ファイル処理
- ○ NumPy

# ファイルパス

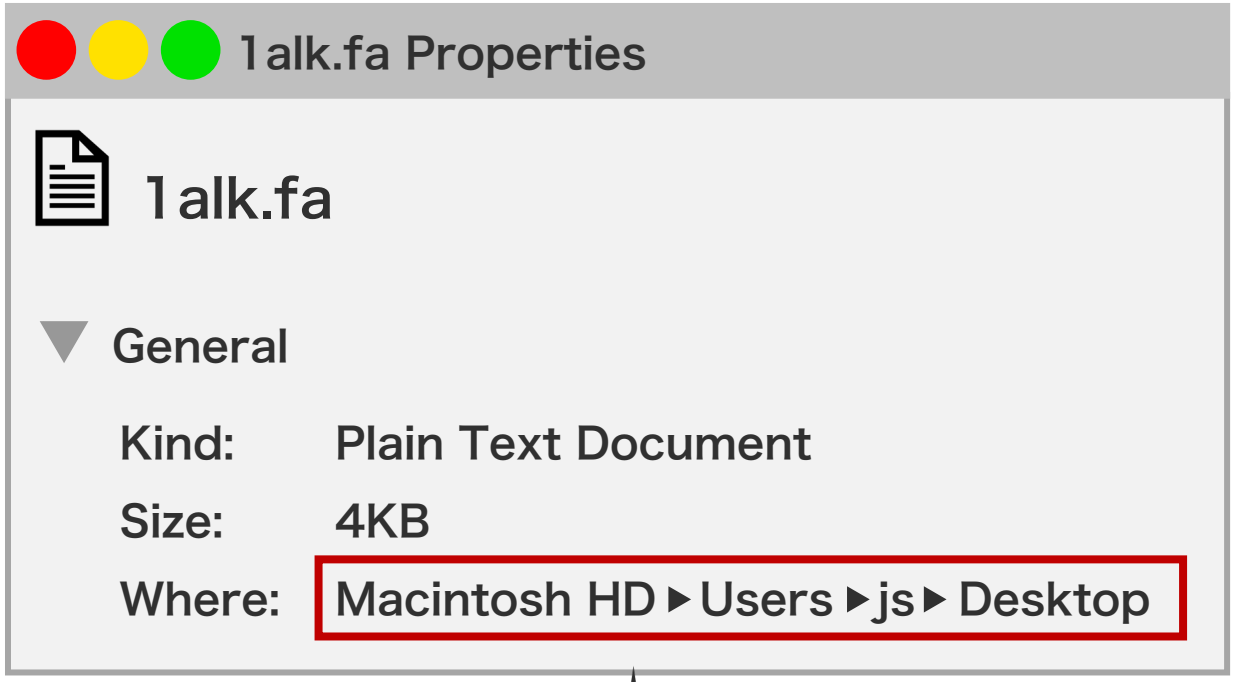

home
prj1
gene.gff
gene.fa
prj2
metadata.txt
data.txt
/home/prj1/gene.fa
/home/prj2/data.txt

# ファイルパス（Macintosh）

1. パスを調べたいファイルを右クリックし、「Get Info」を選ぶ。

2. 「General」タブの「Where」項目にファイルへのパスが記載されている。

   - ファイルパスの「Macintosh HD」は最上層を表し、パスを書くとき「/」に直す。
   - 小さい三角形はフォルダの包含関係を表す。パスを書くときは「/」に直す。




/Users/js/Desktop/1alk.fa

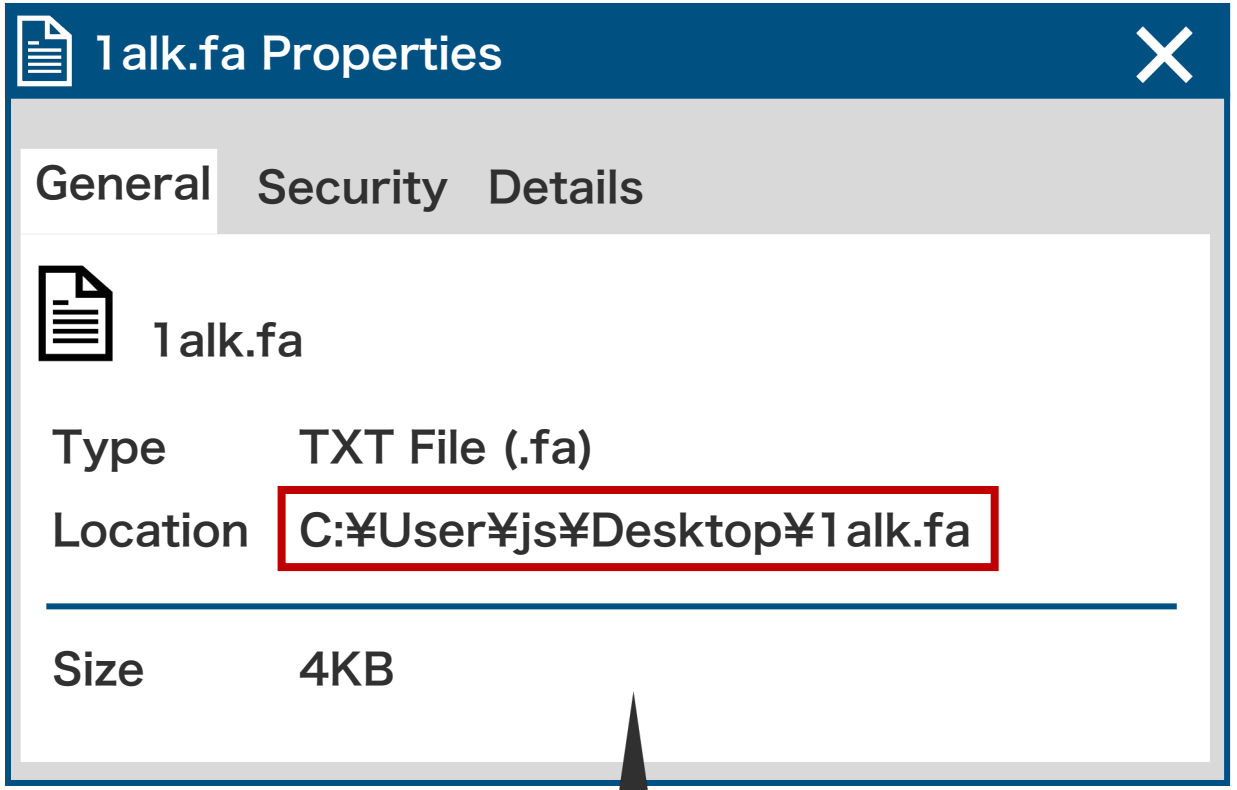
# ファイルパス（Windows）

1. パスを調べたいファイルを右クリックし、「Properties」を選ぶ。

2. 「General」タブの「Location」項目にファイルへのパスが記載されている。

   - **日本語環境の場合、「＼」が「¥」として表示される。**どちらも Windows コンピューター内部では 0x5C として認識されている。

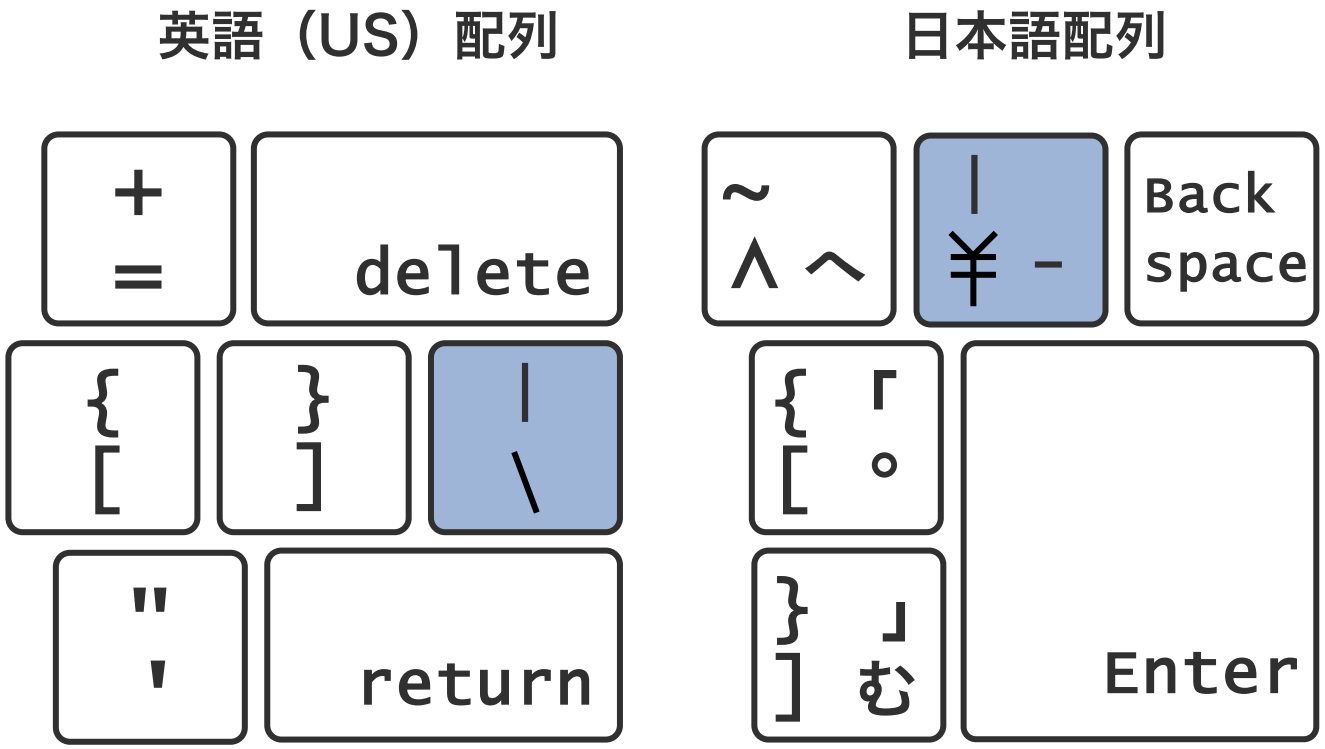
   - パスとして使用するとき、Location 欄に書かれているパス中の**「¥」を「/」に置き換える。**




C:/User/js/Desktop/1alk.fa

# バックスラッシュ（＼）・円マーク（¥）

バックスラッシュは、OS の種類、言語環境や文字コードによって、ディスプレイでの表示が異なる。日本語環境のシステムであれば円マーク「¥」として表示され、それ以外の言語環境では「＼」として表示される。バックスラッシュと円マークは、表示が異なるものの、コンピューター上では同一のコード 0x5C として扱われる。




macOS 日本語環境を使用している場合は、＼ と ¥ の両方を入力できる。Option を押しながら＼または¥キーを押すことで、＼と¥の入力の切り替えができる。

# ファイル

Python でファイルを読み込むには、open 関数を使用する。open 関数に、ファイルへのパスとともにオープン・モードを与えて使う。

| モード | 意味 |
| --- | --- |
| r | 読み込みモード。ファイルが存在しない場合はエラーになる。 |
| w | 書き込みモード。 |
| a | 追記モード。 |

https://aabbdd.jp/notes/data/1alk.fa

```
f = '1alk.fa'


with open(f, 'r') as fh:

    for line in fh:

        print(line)


# >1ALK:A|PDBID|CHAI...
# TPEMPVLENRAAQGNITA...
# TGQYTHYALNKKTGKPDY...
# AHVTSRKCYGPSATSQKC...
```

"IOError: [Errno 2] No such file or directory '1alk.fa'" のようなエラーが起きた場合、ファイルのパスを確認してください。ファイルをダウンロードする際に、"1alk.fa" が勝手に "1alk.fa.txt" に名前変更されることがある。

# ファイルのダウンロード

Google Colab を利用している場合、次のようにすることで、ファイルを直接 Google Golab が実行されているコンピュータ上にダウンロードできる。

```
!wget https://aabbdd.jp/notes/data/1alk.fa

f = '1alk.fa'

with open(f, 'r') as fh:

    for line in fh:

        print(line)


# >1ALK:A|PDBID|CHAI...
# TPEMPVLENRAAQGNITA...
# TGQYTHYALNKKTGKPDY...
# AHVTSRKCYGPSATSQKC...
```

https://aabbdd.jp/notes/data/1alk.fa

# ファイル読み込み

パス f に保存されているファイルを、r モードで開き、ファイルハンドルにセットアップする。

```
f = '1alk.fa'


with open(f, 'r') as fh:

    for line in fh:

        print(line)


# >1ALK:A|PDBID|CHAI...
# TPEMPVLENRAAQGNITA...
# TGQYTHYALNKKTGKPDY...
# AHVTSRKCYGPSATSQKC...
```

# ファイル読み込み

ファイルの内容がリストに変換されてファイルハンドルに代入される※。ファイルの i 行目の文字列情報が、リストの i 番目の要素に代入されている。そのため、for 構文を利用して、リストの先頭から要素を順に取り出せば、ファイルの内容を 1 行目から順に取り出せるようになる。

```
>1ALK
TPEMPV
TGQYTH
AHVTSR
```

```
fh = [
  '>1ALK',
  'TPEMPV',
  'TGQYTH',
  'AHVTSR'
]
```

▶

```
f = '1alk.fa'

with open(f, 'r') as fh:
    for line in fh:
        print(line)

# >1ALK:A|PDBID|CHAI...
# TPEMPVLENRAAQGNITA...
# TGQYTHYALNKKTGKPDY...
# AHVTSRKCYGPSATSQKC...
```

※ 正確には、ファイルの何行の何文字目まで読み込んだかの位置情報（ファイルポインター）がファイルハンドルに保存される。ファイルハンドルに対して for 構文を適用すると、ポインターを自動的に 1 行ずつ進めさせることができる。また、read メソッドと for 構文を組み合わせることで、ポインターを 1 文字ずつ進めさせることもできる。

# ファイル読み込み

ファイルの内容がリストに変換されてファイルハンドルに代入される※。ファイルの i 行目の文字列情報が、リストの i 番目の要素に代入されている。そのため、for 構文を利用して、リストの先頭から要素を順に取り出せば、ファイルの内容を 1 行目から順に取り出せるようになる。

```
f = '1alk.fa'

with open(f, 'r') as fh:
    for line in fh:
        print(line)

# >1ALK:A|PDBID|CHAI...
# TPEMPVLENRAAQGNITA...
# TGQYTHYALNKKTGKPDY...
# AHVTSRKCYGPSATSQKC...
```

```
>1ALK
TPEMPV
TGQYTH
AHVTSR
```

```
fh = [
  '>1ALK\n',
  'TPEMPV\n',
  'TGQYTH\n',
  'AHVTSR\n'
]
```

改行コード

※ 正確には、ファイルの何行の何文字目まで読み込んだかの位置情報（ファイルポインター）がファイルハンドルに保存される。ファイルハンドルに対して for 構文を適用すると、ポインターを自動的に 1 行ずつ進めさせることができる。また、read メソッドと for 構文を組み合わせることで、ポインターを 1 文字ずつ進めさせることもできる。

# ファイル読み込み

```
f = '1alk.fa'


with open(f, 'r') as fh:

    for line in fh:

        print(line)


# >1ALK:A|PDBID|CHAI...
# TPEMPVLENRAAQGNITA...
# TGQYTHYALNKKTGKPDY...
# AHVTSRKCYGPSATSQKC...
```

すべての行の読み込みが終わり、ファイルが閉じられる。 ▶

# エスケープシーケンス

Python では、特殊な用途向けに、いくつかの特殊文字が定義されている。例えば、改行とタブがその特殊文字にあたる。改行文字は、人には見えないが、コンピュータが改行として認識するために必要な特殊文字である。限られたアルファベットと数字の中で、このような特殊文字を表すためには、既存文字を組み合わせて表現する。一般に、バックスラッシュに 1 文字を足して、特殊文字を表現していることが多い。例えば、改行ならば \n、タブならば \t のように表現する。

```
s = 'Smile. nTomorrow will be worse.'
s
# 'Smile. nTomorrow will be worse.'



s = 'Smile. \nTomorrow will be worse.'
s
# 'Smile.'
# 'Tomorrow will be worse.'



s = 'Smile. \\nTomorrow will be worse.'
s
# 'Smile. \nTomorrow will be worse.'
```

# 問題 F1-1

リスト fh の要素のうち、「>」から始まる要素の個数を求めよ。

```
fh = ['>1ALK:A',
      'TPEMPVL',
      'TGQYTHA',
      '>1ALK:B',
      'TPEMPVL',
      'TGQYTHA']
```

# 問題 F1-2

1alk.fa ファイルを読み込んで、「>」から始まる行の行数を求めよ。

```
f = '1alk.fa'
n = 0
```

https://aabbdd.jp/notes/data/1alk.fa

# 問題 F1-3

ft.fa ファイルは FASTA 形式のテキストファイルである。このファイルは「>」から始まる行に遺伝子名が記載され、それ以降の行に遺伝子の塩基配列が記載されている。ft.fa に記載されいてる遺伝子の塩基配列の長さを求めよ。

```
f = 'ft.fa'
l = 0
```

小文字エル l、大文字アイ I、数字 1 はフォントによって区別しにくい場合がある。

https://aabbdd.jp/notes/data/ft.fa

# 問題 F1-4

ft.fa に記載された塩基配列について、A の出現確率を求めよ。

```
f = 'ft.fa'
pA = 0
```

https://aabbdd.jp/notes/data/ft.fa

# 問題 F1-5

diversity_galapagos.txt は、タブ区切りのテキストファイルであり、ガラパゴス島における種の多様性データが記載されている。このファイルを読み込み、面積（Area 列）の最も大きい島の名前（Island 列）を答えよ。ただし、このテキストファイルには、「#」から始まるコメント行とデータの属性を示すヘッダー行が含まれていることに注意。

```
f = 'diversity_galapagos.txt'
island_name = ''
```

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

# 問題 F1-6

diversity_galapagos.txt は、タブ区切りのテキストファイルであり、ガラパゴス島における種の多様性データが記載されている。面積（Area）あたりの種数（Species）が、最も大きい大きい島の名前とそのときの面積当たりの種数を求めよ。

```
f = 'diversity_galapagos.txt'
island_name = ''
island_dens = 0
```

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

# ファイル書き込み

ファイルの書き込みには w および a モードが定義されている。このほかにも w+ や a+ などのモードも定義されているが、初めは w モードのみ使えればよい。

| モード | 意味 |
| --- | --- |
| r | 読み込みモード。ファイルが存在しない場合はエラーになる。 |
| w | 書き込みモード。ファイルが存在しない場合は新規作成される。ファイルが存在する場合は、既存のファイルを削除したうえで新規作成する。 |
| a | 追記モード。既存のファイルの最後に追記する。ファイルが存在しない場合は新規作成される。 |

```
f = 'output.fa'


with open(f, 'w') as outfh:

    outfh.write('abcdefg')
    outfh.write('1234567')
```

# ファイル書き込み

パス f を書き込みモードで開き、ファイルハンドル ▶ にセットアップする。

```
f = 'output.fa'


with open(f, 'w') as outfh:

    outfh.write('abcdefg')
    outfh.write('1234567')
```

# ファイル書き込み

abcdefg をファイルに書き込む ▶

```
f = 'output.fa'


with open(f, 'w') as outfh:

    outfh.write('abcdefg')
    outfh.write('1234567')
```

# ファイル書き込み

1234567 をファイルに書き込む ▶

```
f = 'output.fa'


with open(f, 'w') as outfh:

    outfh.write('abcdefg')
    outfh.write('1234567')
```

# ファイル書き込み

書き込みが終了し、ファイルが閉じられる。 ▶

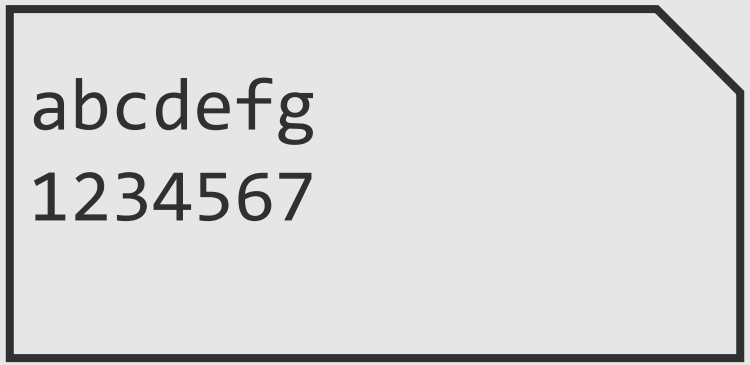
```
f = 'output.fa'


with open(f, 'w') as outfh:

    outfh.write('abcdefg')
    outfh.write('1234567')
```

```
abcdefg1234567
```

# ファイル書き込み

書き込みが終了し、ファイルが閉じられる。 ▶

```
f = 'output.fa'


with open(f, 'w') as outfh:

    outfh.write('abcdefg\n')
    outfh.write('1234567\n')
```



# 問題 F1-7

ft.fa ファイルは FASTA 形式のテキストファイルである。このファイルは「>」から始まる行に、塩基配列の名前が記載され、それ以降の行は塩基配列が大文字で記載されている。配列の名前を変更しないで、塩基配列をすべて小文字に変換し、これらの情報を新しいファイル new_ft.fasta に保存するプログラムを作成せよ。

```
>ft
ACCGTGFA
ATTTGCTA
CACATTTA
AAAC
```

```
>ft
accgtgfa
atttgcta
cacattta
aaac
```

https://aabbdd.jp/notes/data/ft.fa

```
f = 'ft.fa'
l = 0
```

# 農学生命情報科学特論Ⅰ

② 

- テキスト処理
- ファイル処理
- NumPy

# 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。

```
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])
```

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。

# 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])
```

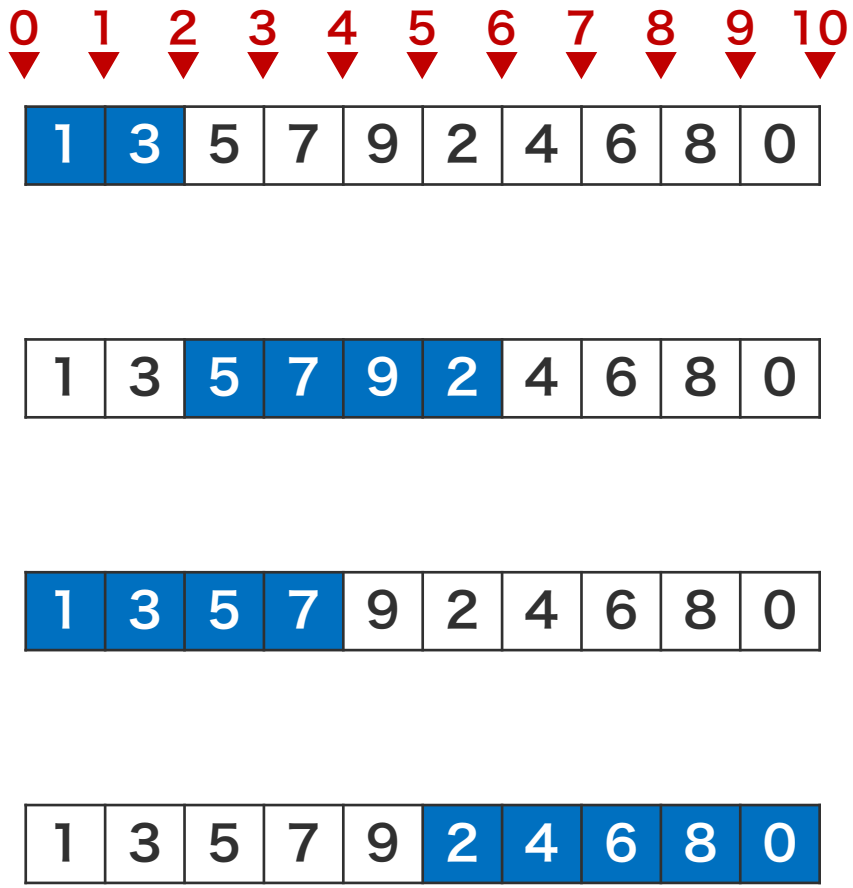
# 1 次元配列 / スライス

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])
```

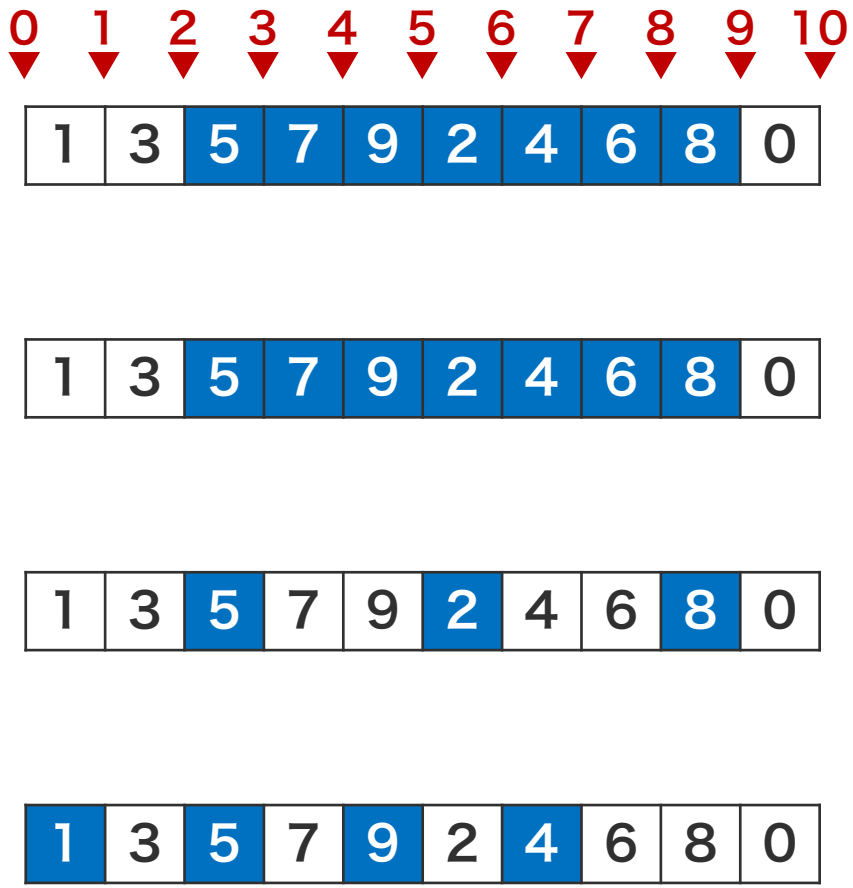
```
a[5:]
# np.array([2, 4, 6, 8, 0])
```

# 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])
```

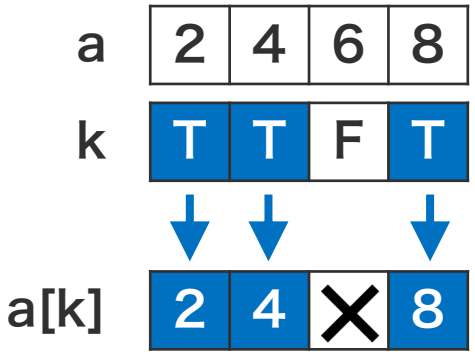
# 1 次元配列 / フィルター

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]
```

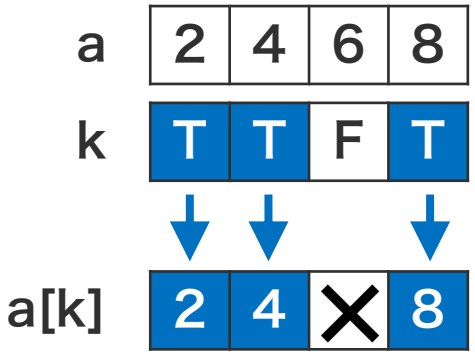
# 1 次元配列 / フィルター

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。

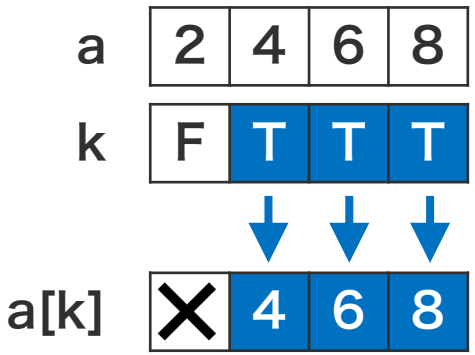



```
import numpy as np


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])
```

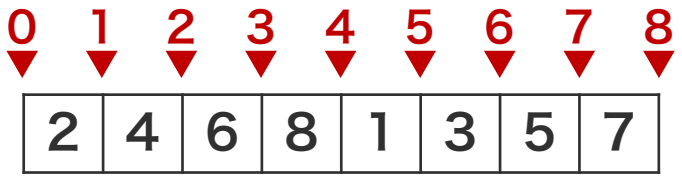
フィルター k を用いて、ベクトル a の要素をフィルタリングしているイメージ。

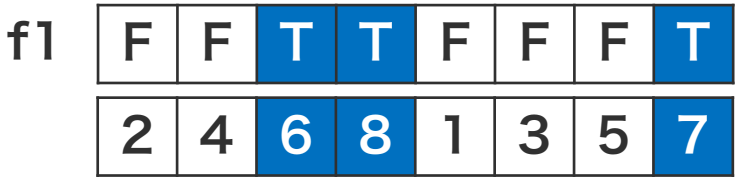
# 1 次元配列 / フィルター

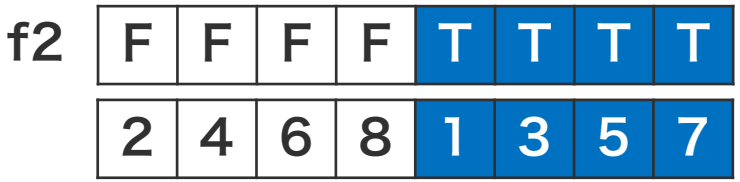
フィルターは、True または False を組み合わせて直打ちで作成できるが、ある条件を制定して、その条件に基づいて作成することが一般的である。例えば、5 よりも大きい値を取り出すフィルターや奇数の値を取り出すフィルターは、右のように作成する。

```
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

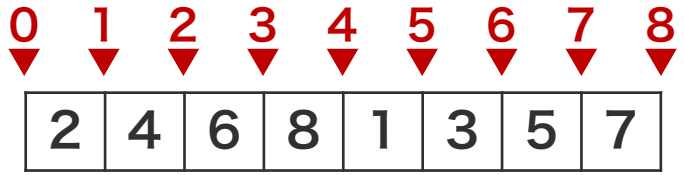


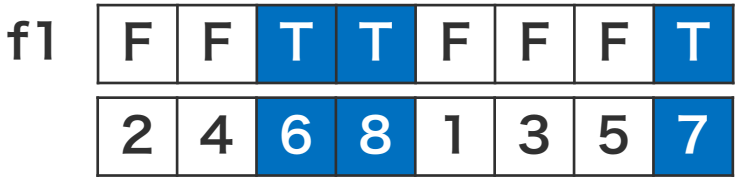



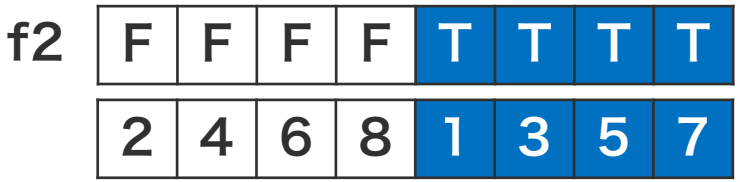



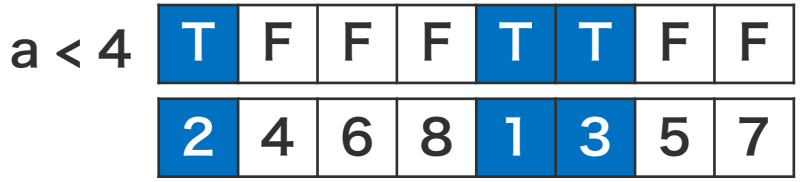

```
import numpy as np


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])
```

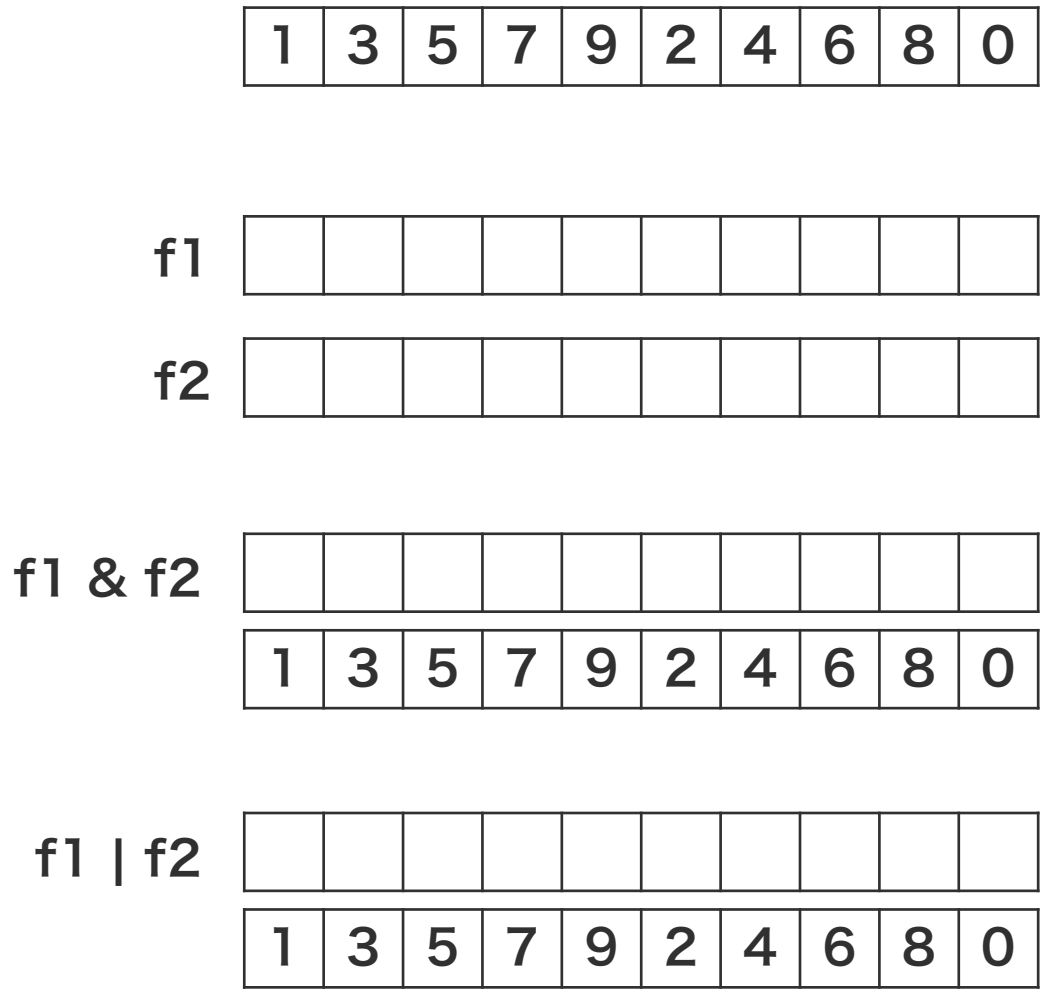
フィルターを一次変数に保存せずに、直接使用することもできる。

# 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。

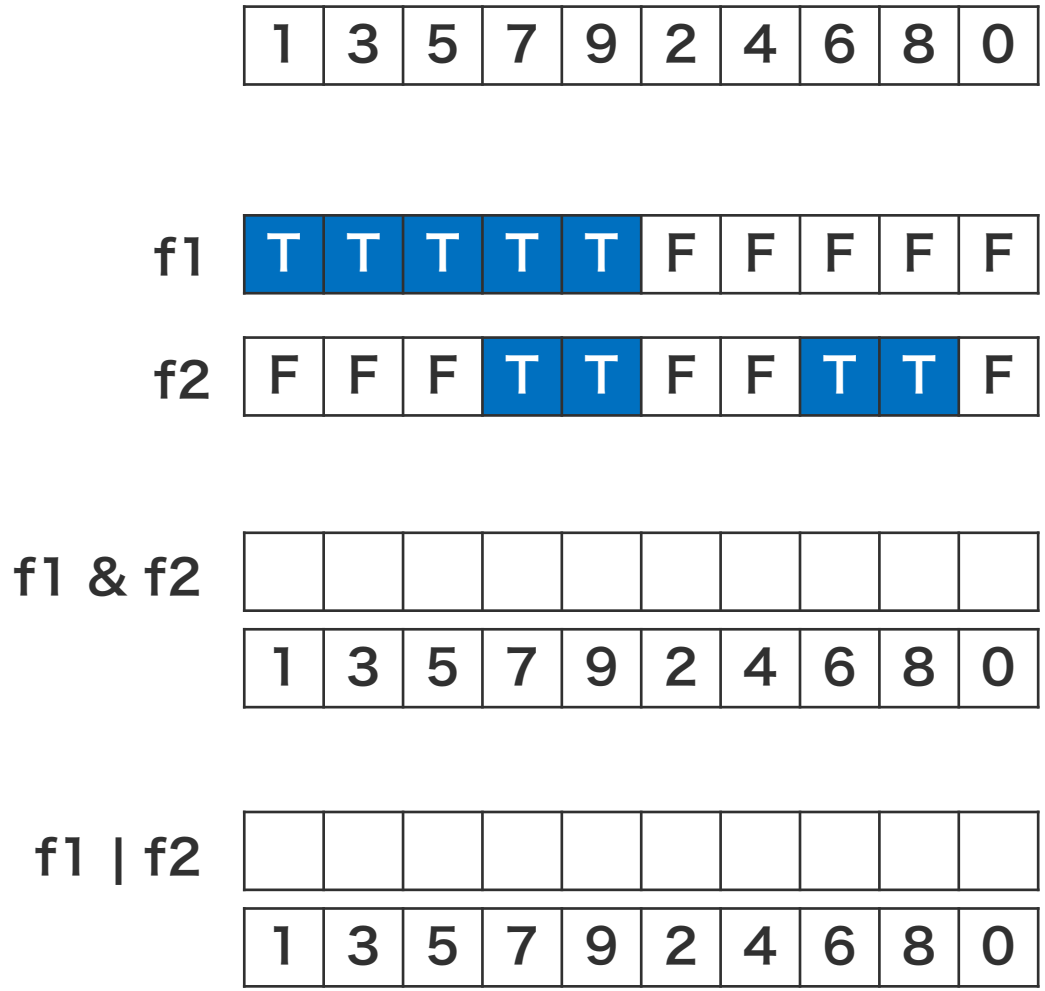
```
import numpy as np

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



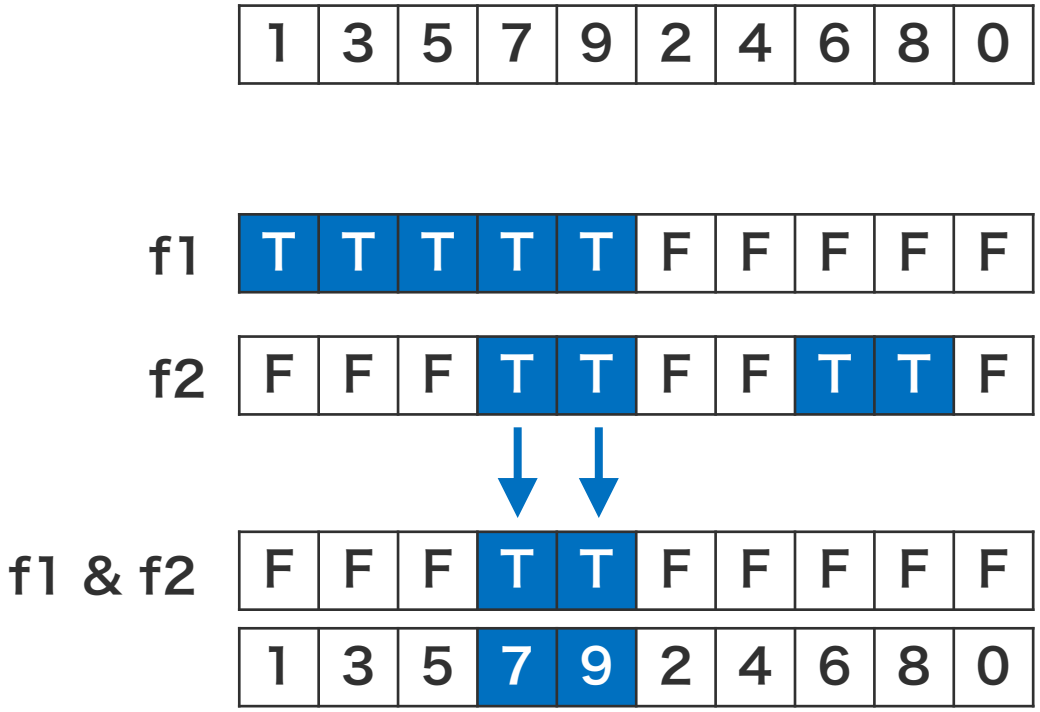

```
import numpy as np

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



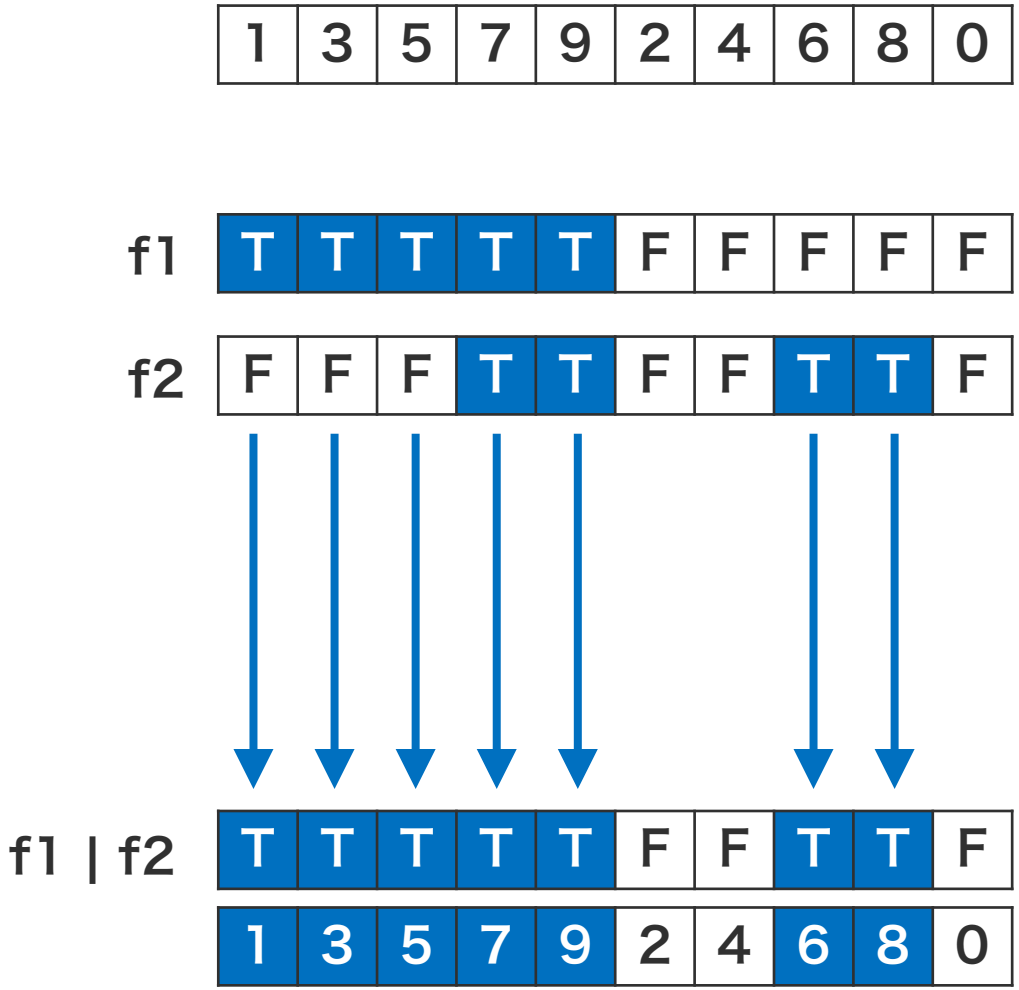

```
import numpy as np

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

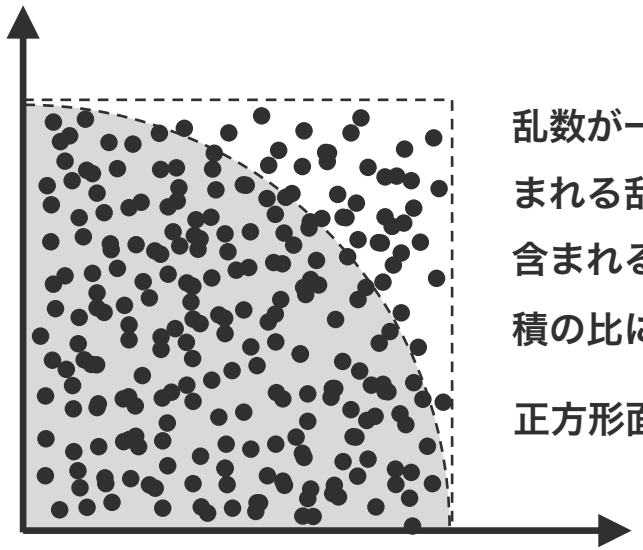
NumPy の random モジュールには擬似乱数を生成する機能が実装されている。次の表に、一様分布および正規分布から乱数を生成する関数を示した。これ以外にも、ポアソン分布やガンマ分布などの分布から乱数を生成する関数が多数用意されている。必要なときに調べて使うとよい。

| 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 : $\pi/4$

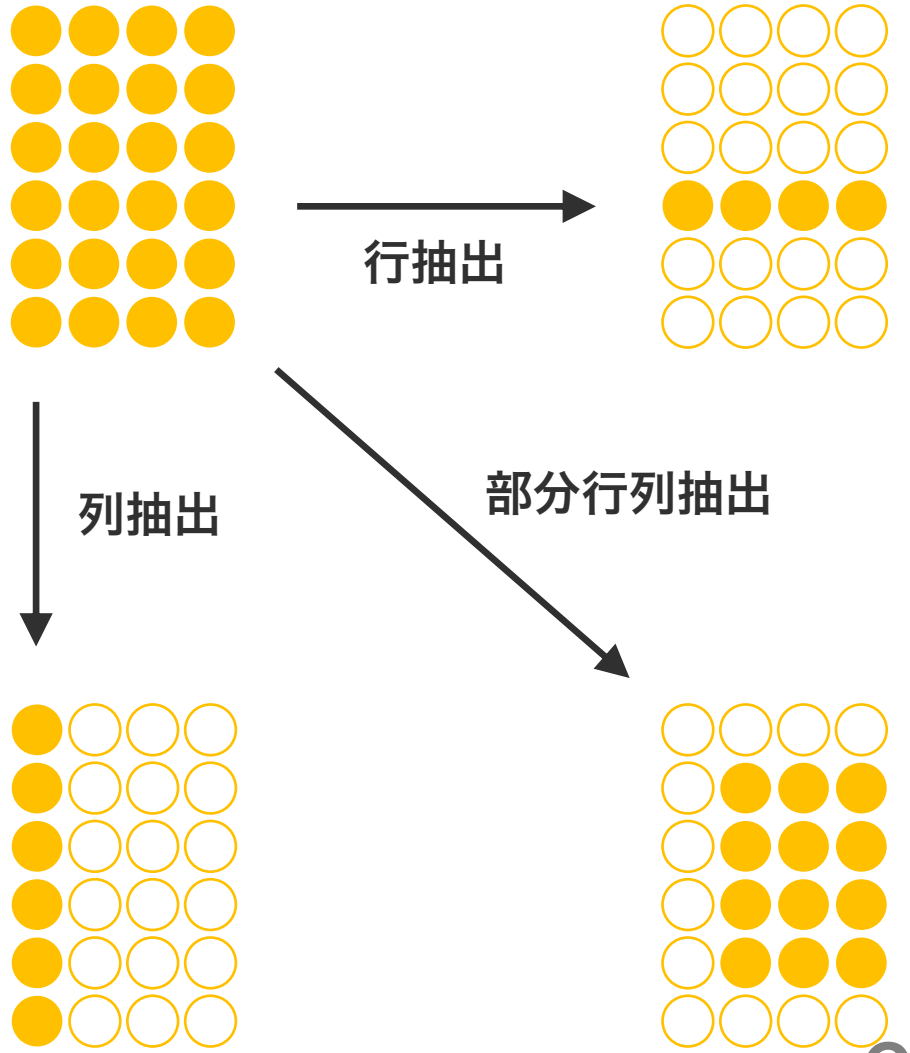
```
import numpy as np

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

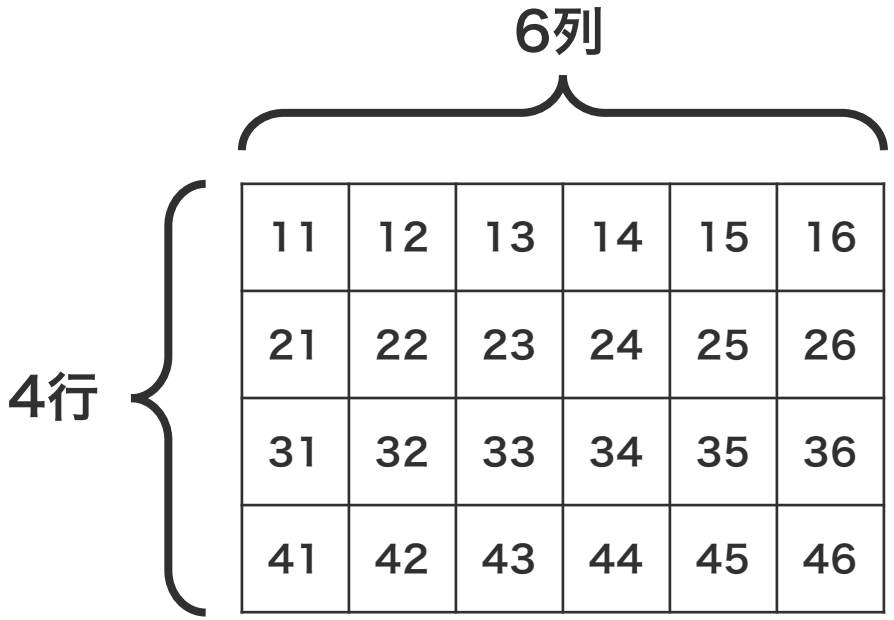
２次元配列の場合、np.identity 関数を使用して、単位行列を作成することができる。

```
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。



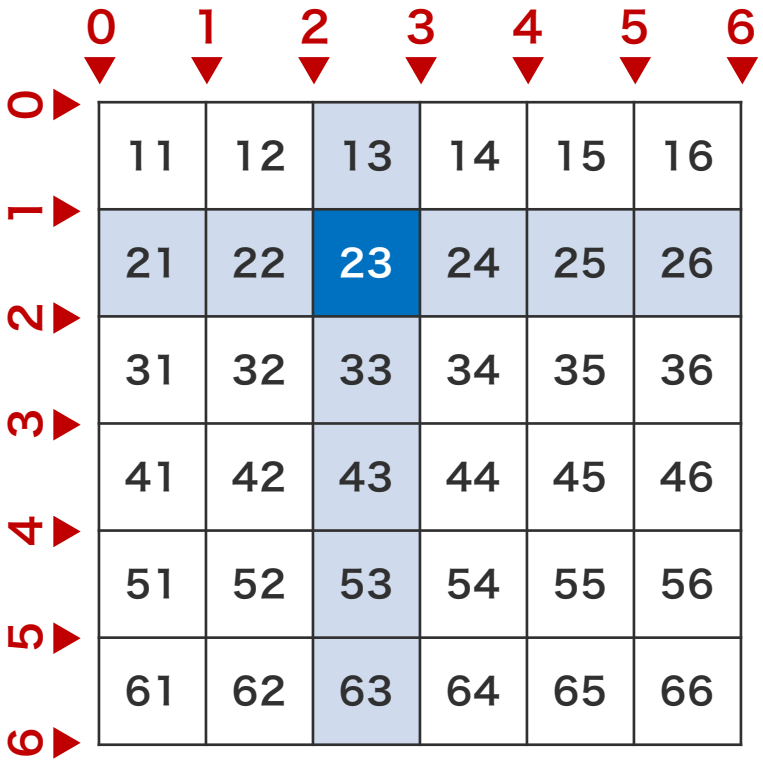

```
a = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45,
46]])


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。

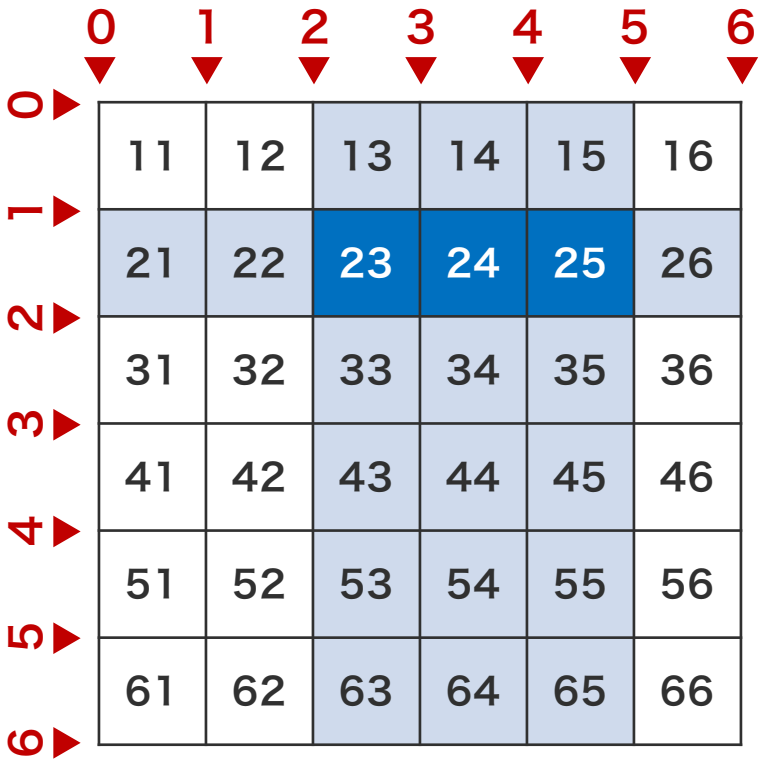
```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65,
66]])

b[1, 2]


b[1, 2:5]


b[1:5, 3]


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

# 2 次元配列



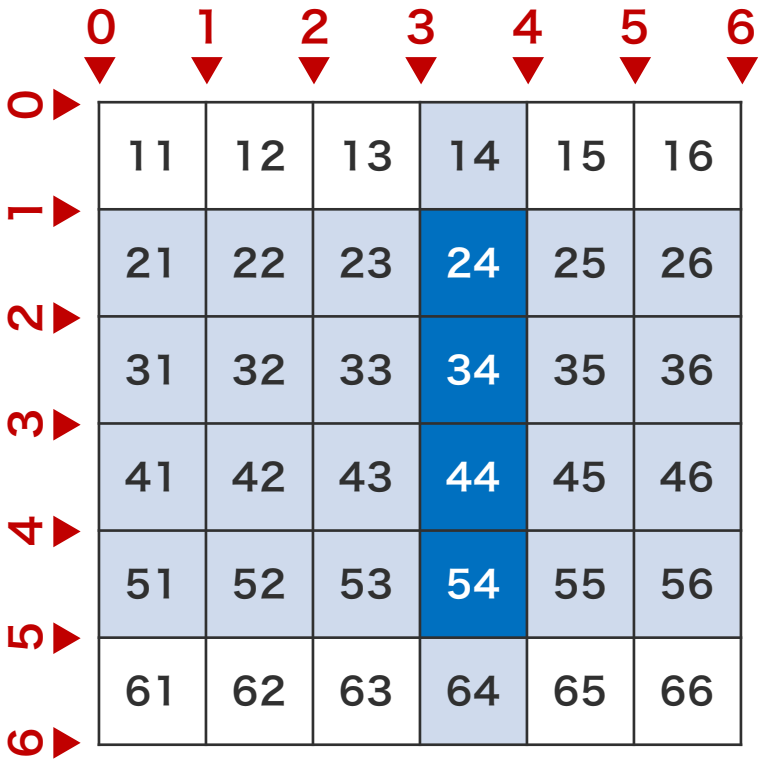

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65,
66]])

b[1, 2]
# 23

b[1, 2:5]


b[1:5, 3]


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

# 2 次元配列



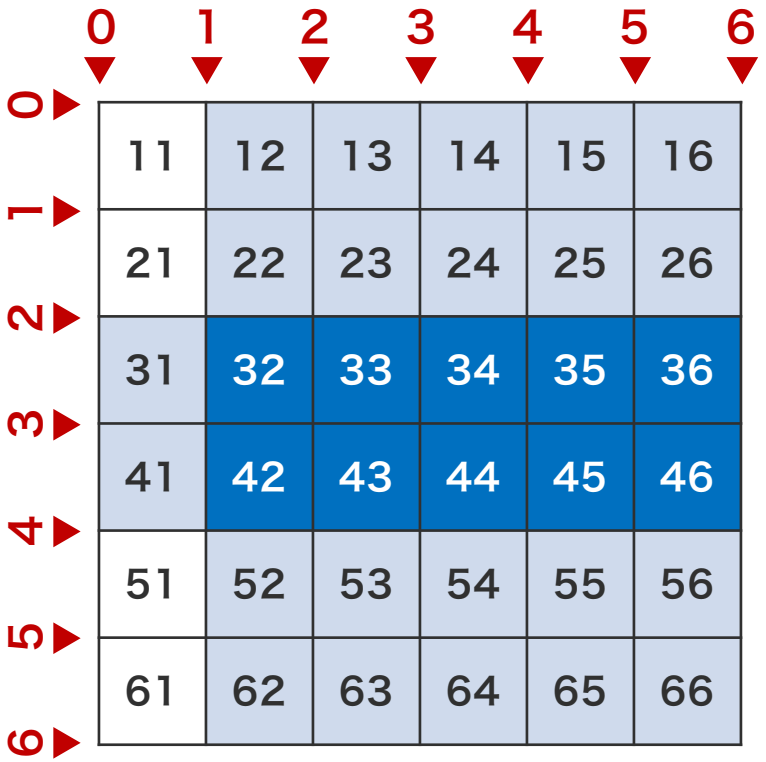

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65,
66]])

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



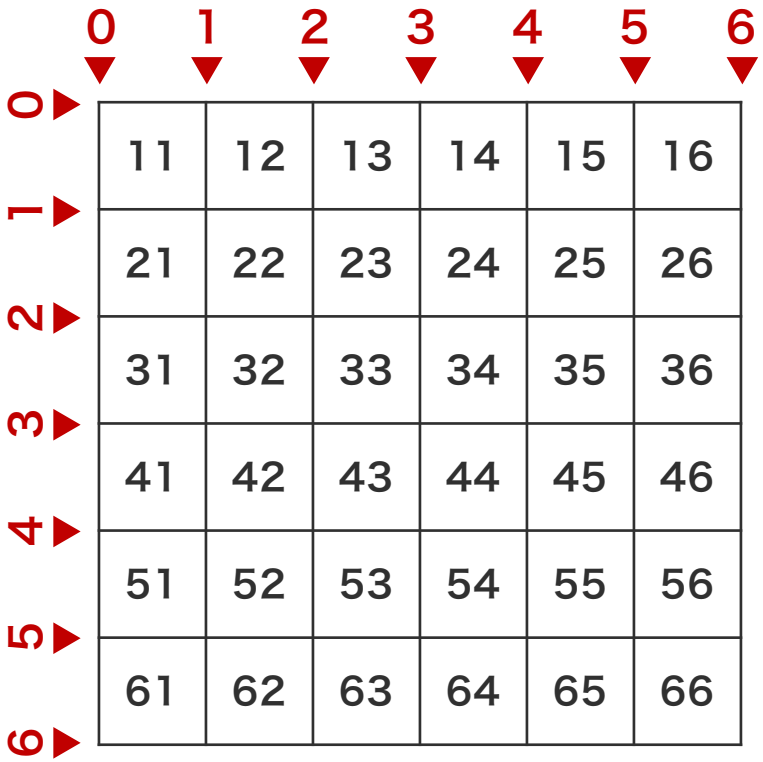

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65,
66]])

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



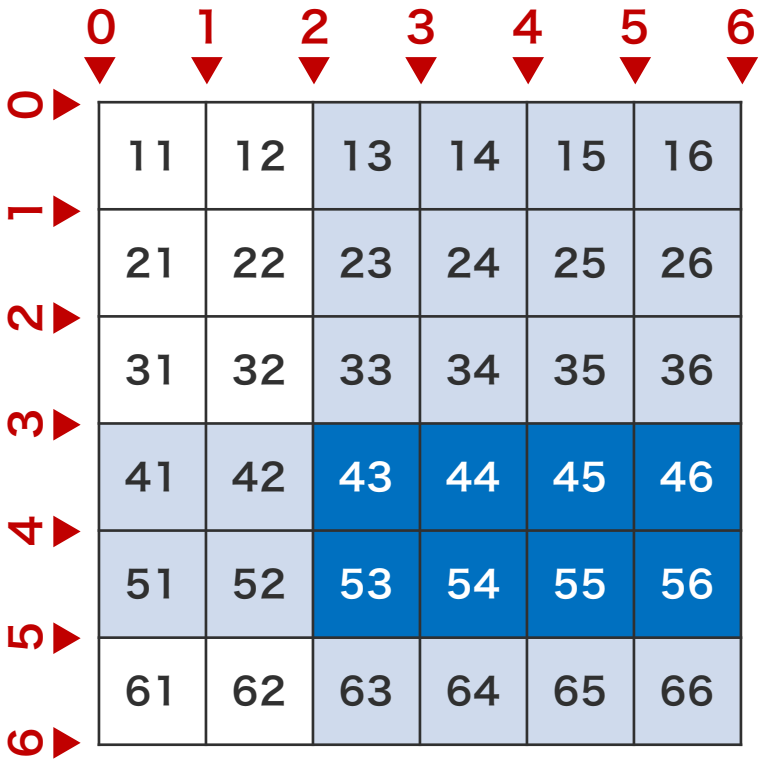

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65,
66]])

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



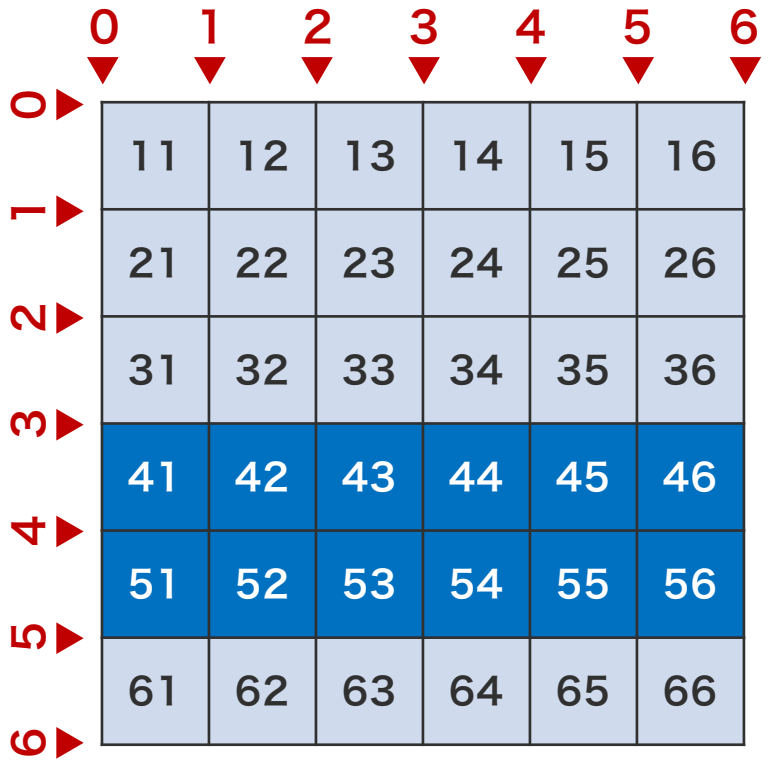

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65,
66]])

b[3:5, 2:]


b[3:5, :]


b[:, 2]


b[:, 1:3]
```

# 2 次元配列



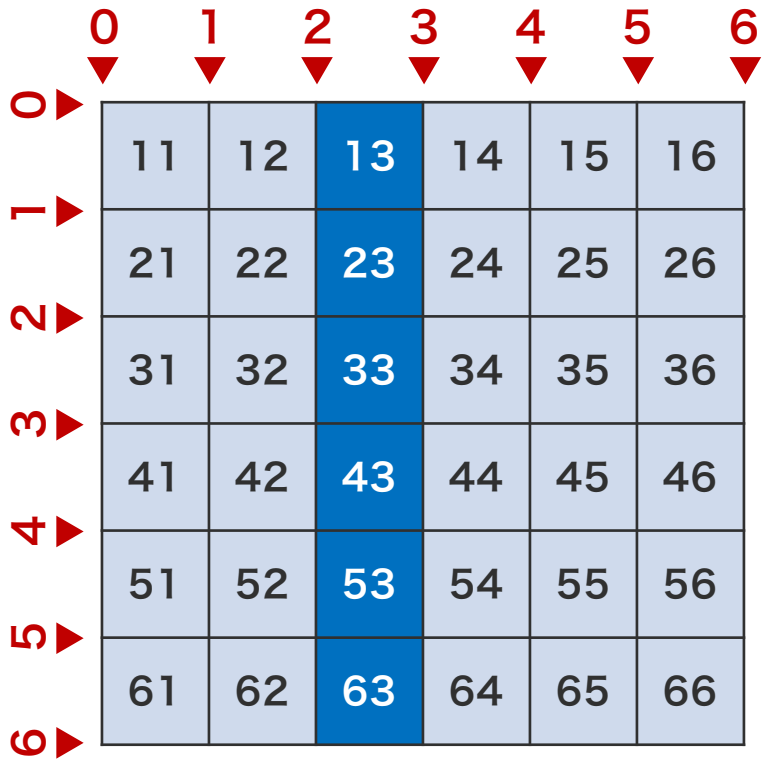

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65,
66]])

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



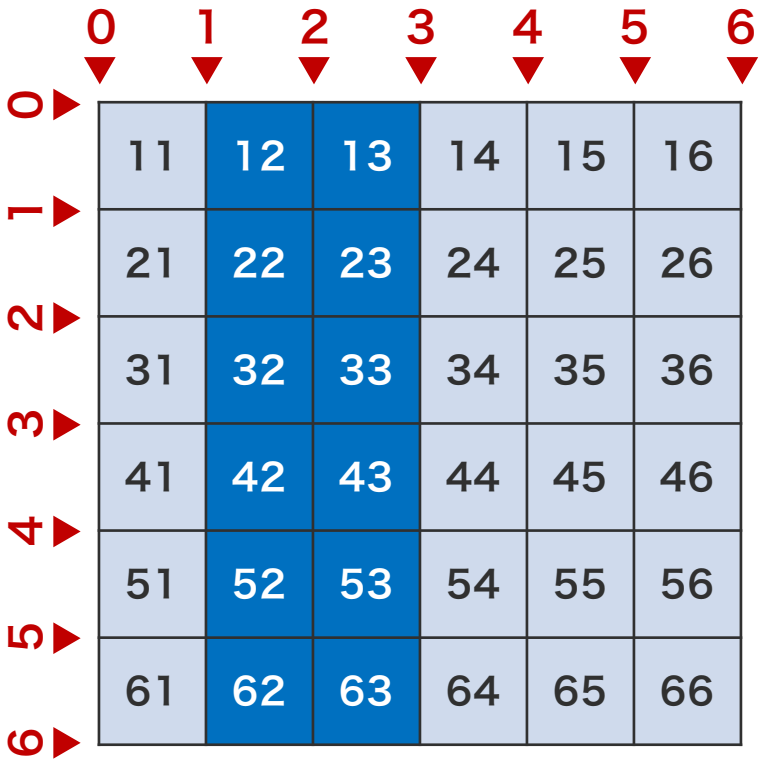

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65,
66]])

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



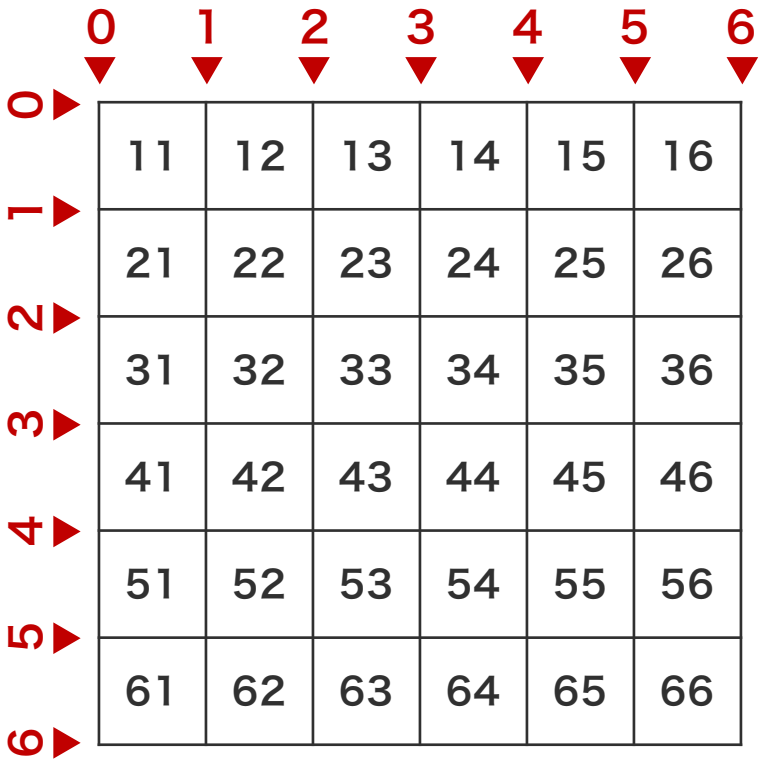

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65,
66]])

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



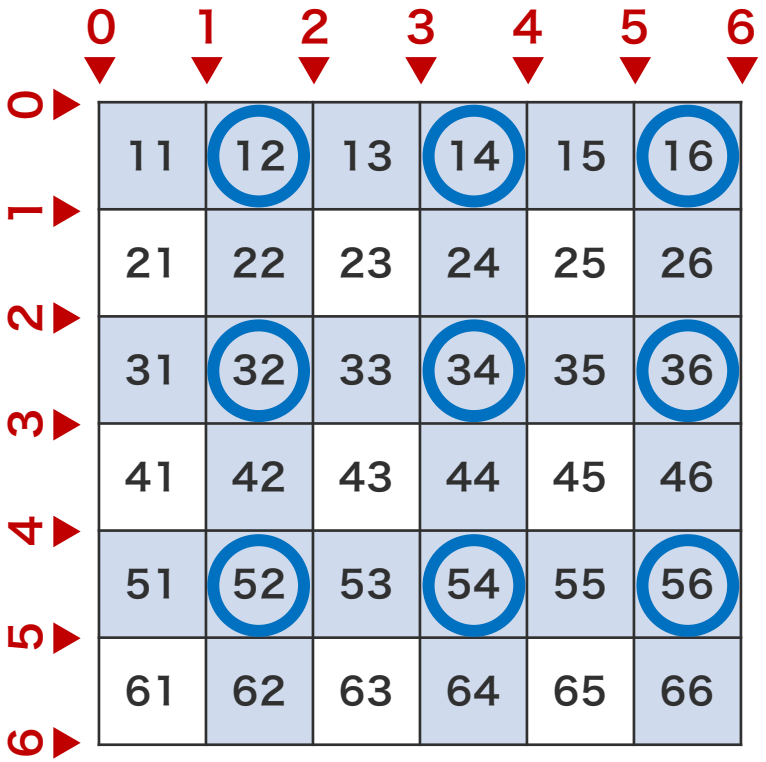

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65,
66]])

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



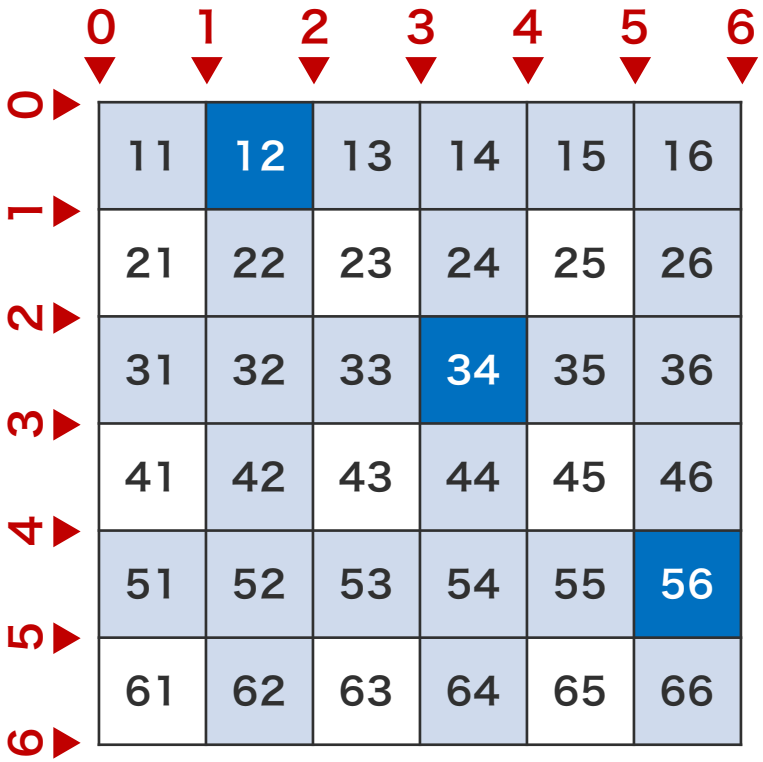

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65,
66]])

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

b[r, c]
```

# 2 次元配列 / 注意点



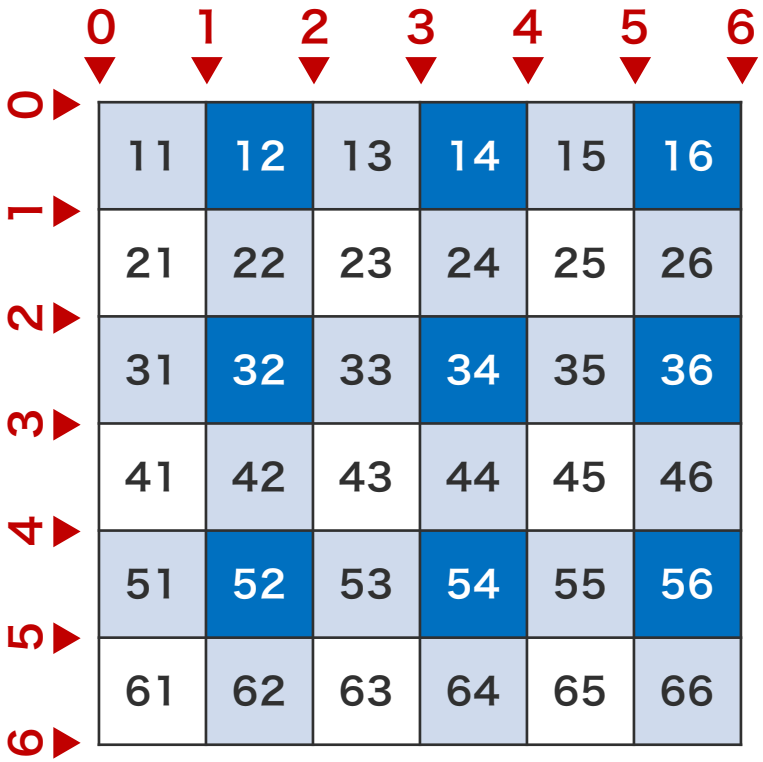

```
b = np.array([[11, 12, 13, 14, 15, 16],
              [21, 22, 23, 24, 25, 26],
              [31, 32, 33, 34, 35, 36],
              [41, 42, 43, 44, 45, 46],
              [51, 52, 53, 54, 55, 56],
              [61, 62, 63, 64, 65,
66]])

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

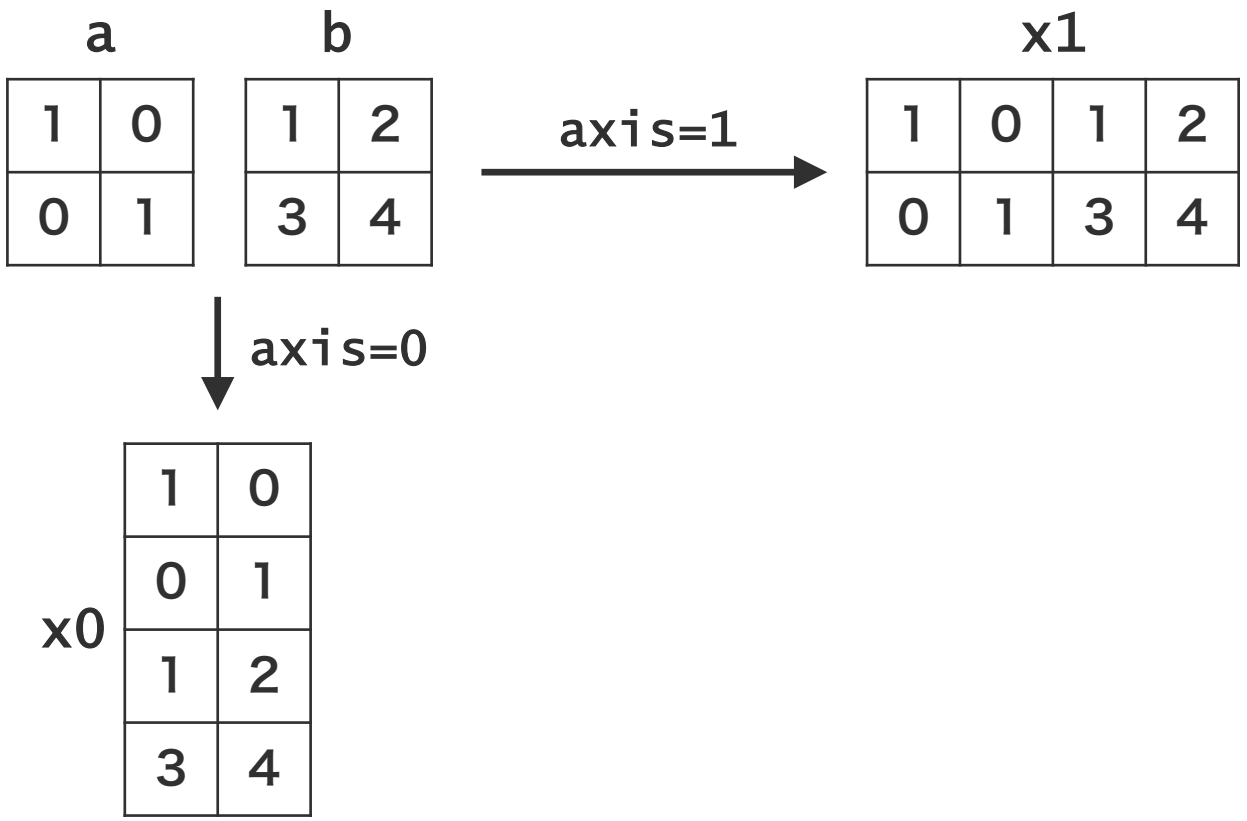
NumPy の 2 次元配列に、行列演算が定義されている。配列の各要素同士の加減乗除の他に、内積や外積も簡単に求めることができる。

| 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]])
```

# 行列操作 / np.concatenate

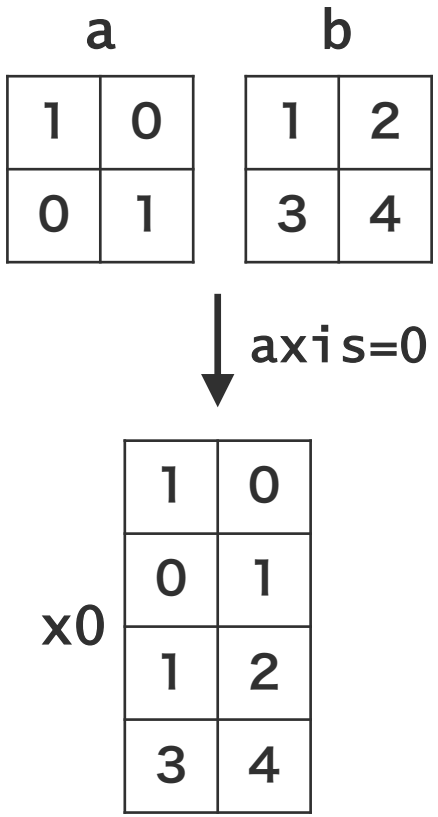
np.concatenate 関数は、複数のNumPy 配列を結合する機能を持つ。この関数を使うとき、結合する軸（次元）方向を axis 引数で指定。省略された場合は axis=0 となる。




```
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

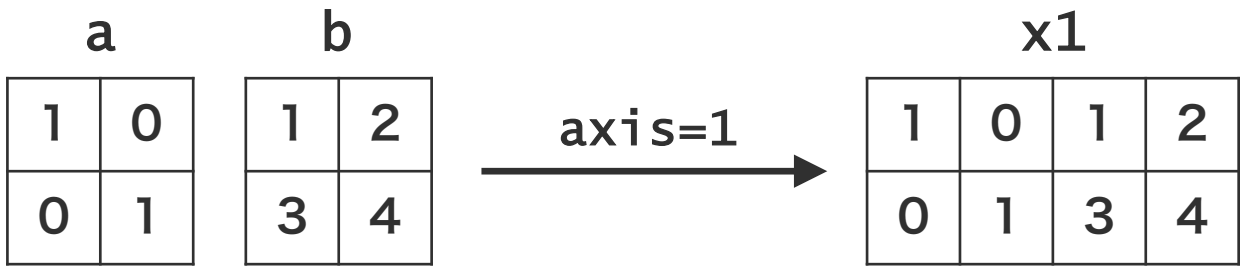
np.vstack 関数は複数の配列を第 1 次元方向に結合する機能を持つ。axis=0 のときの np.concatenate 関数の機能と同じ。




```
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]]])
```

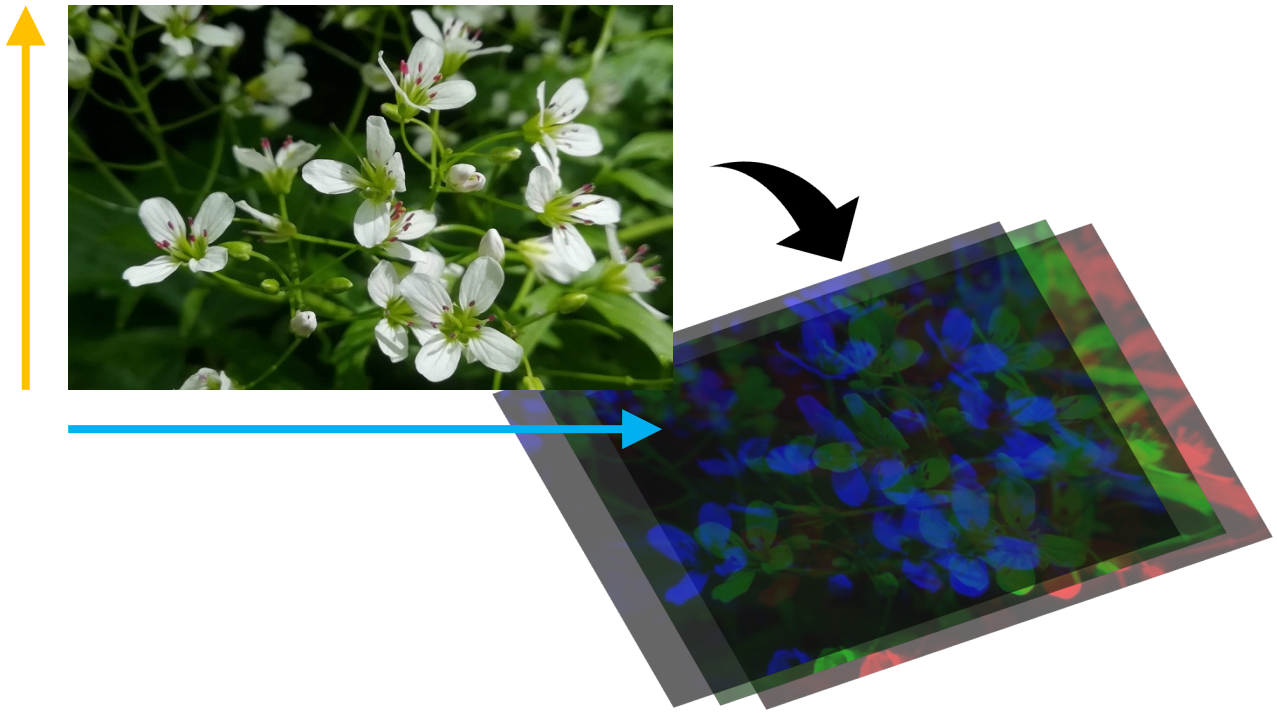
a は 12 要素。最初の 2 次元に 2 要素ずつを配置した場合、最後の次元は自動的に 3 になる。

# 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

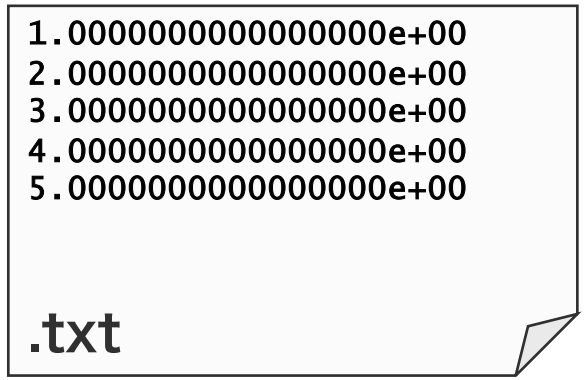
NumPy のバイナリデータを保存しているファイルからデータを読み込むときに load 関数を使用する。複数の配列を含むファイルを読み込むときも load 関数を使用するが、その際、複数個の配列はディクショナリに似たデータ構造でオブジェクトに代入される。そのオブジェクトの .files 属性には、配列の名前の一覧が保存されている。

```
# 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.])
```