



2022/10/25

スライド作成者：中内

GCI2022WINTER  
WEEK2

配列データ分析のための  
Numpy

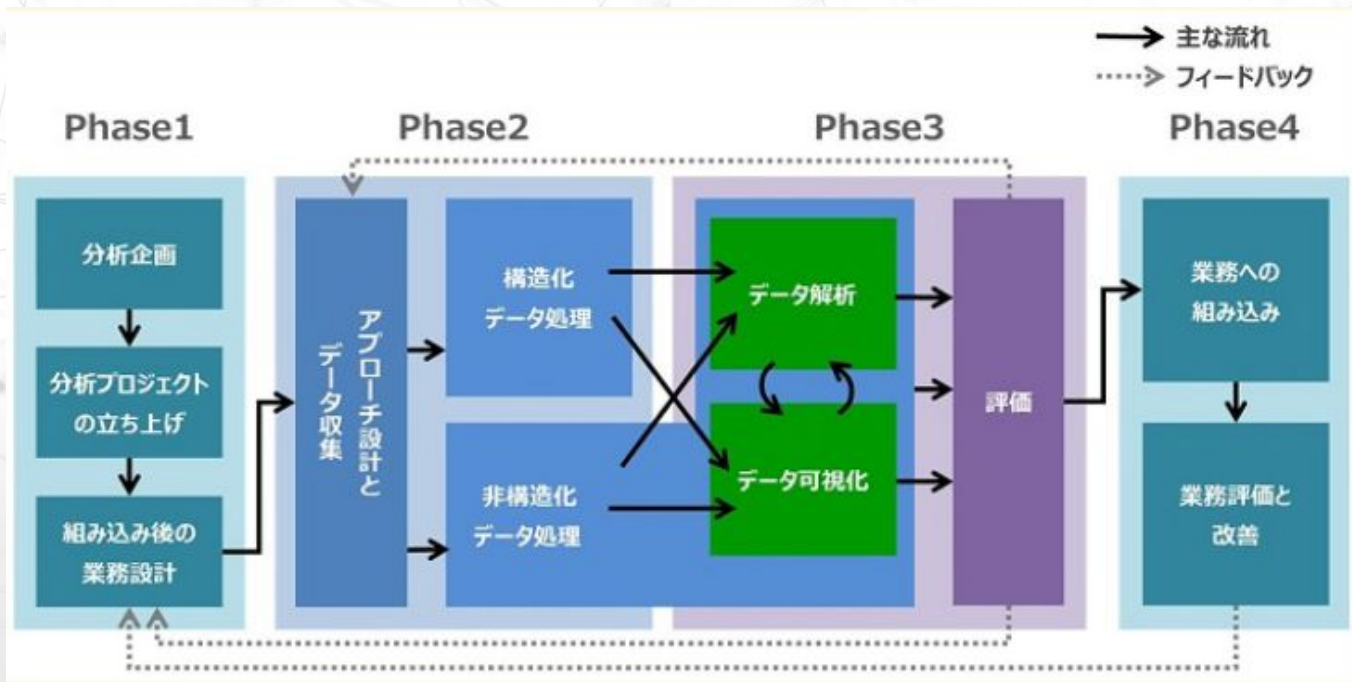
# 「Numpy」とは

元々数値計算用に設計されたわけではなかった  
Pythonで科学計算を容易くするために開発された  
「ライブラリ」

# Numpyを使う利点

- `numpy.ndarray`というn次元配列データを取扱うための強力なデータ形式を使える
- 科学計算の豊富な関数やメソッド
- 計算が高速(  
numpy内部は高速なC言語が主に使われており、  
また最適化された計算ロジックで実装されている
- forループを使った動的な処理にも向く

# データサイエンスプロジェクトの流れ



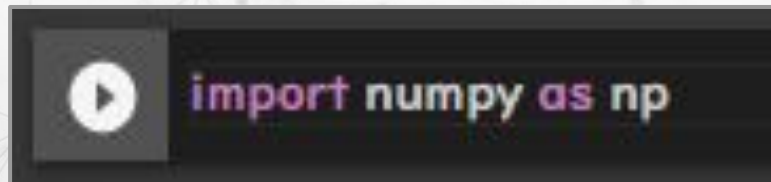


The background of the slide is a complex network diagram. It consists of numerous small, light-gray circular nodes scattered across the frame. These nodes are interconnected by a dense web of thin, light-gray lines, creating a mesh-like structure that resembles a neural network or a data connectivity map. The overall aesthetic is technical and modern.

# Numpyの使い方

# Numpyの基本的な使い方

- Numpyをimportするときは「**np**」を略称とするのが慣習
- (※「as○○」で略称を設定する)



# Numpyの基本的な使い方

`np.○○○( { 処理対象のオブジェクト } )`

様々な関数名を`np.`の後ろに指定して`()`内のオブジェクトに処理を適用する



```
list_A = [0, 1, 2]  
array_A = np.array(list_A)
```

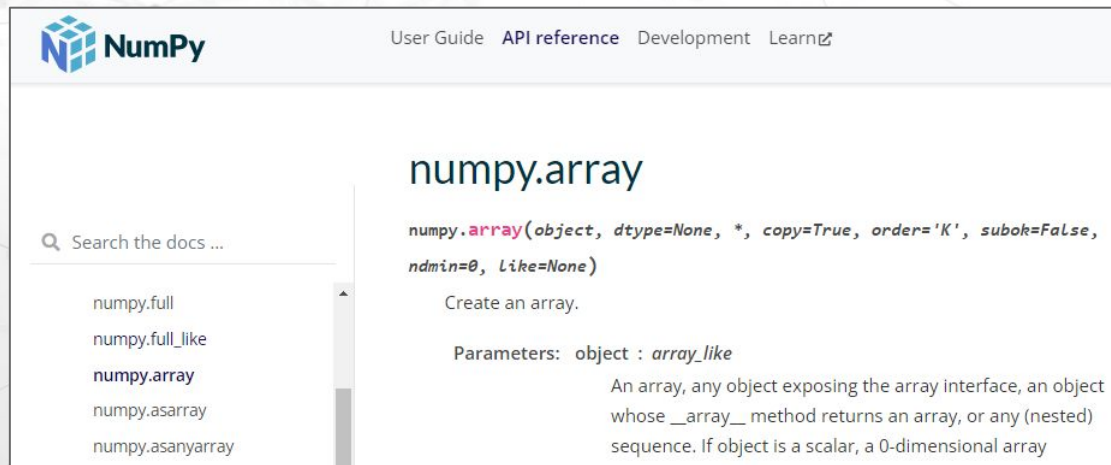


```
[0 1 2]  
numpy.ndarray
```

リスト形式のデータが格納された変数「list\_A」に対し、`np.array()`というnumpyのメソッドを適用した様子

# Numpyの基本的な使い方

- 関数は膨大な種類が用意されている。
- 暗記するものではなく、調べて使うもの。



The screenshot shows the NumPy User Guide API reference page for `numpy.array`. The page has a light blue header with the NumPy logo and navigation links: "User Guide", "API reference", "Development", and "Learn". On the left, there is a search bar labeled "Search the docs ..." and a list of search results: `numpy.full`, `numpy.full_like`, `numpy.array` (highlighted), `numpy.asarray`, and `numpy.asanyarray`. The main content area displays the title `numpy.array` in a large font, followed by the function signature: `numpy.array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0, like=None)`. Below the signature, it says "Create an array." and "Parameters: object : array\_like". A detailed description follows: "An array, any object exposing the array interface, an object whose `__array__` method returns an array, or any (nested) sequence. If object is a scalar, a 0-dimensional array".



The background of the slide features a complex network of thin, light gray lines connecting numerous small, dark gray dots. These dots are scattered across the entire frame, creating a web-like or molecular structure that suggests connectivity and data flow.

# **numpy.ndarray**

## というデータ型を知る

その前に復習: Pythonにはリスト型というデータ形式がある

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

**[ ]** (角カッコ、スクエアブラケット) で囲まれ、  
カンマで区切られた複数の値を持つ形式

# numpy.ndarrayとはn次元配列データ

axis = 1

	[0]	[1]	[2]
[0]	70	95	80
[1]	35	42	50
[2]	76	37	65

axis = 0

- ndarrayはリスト型を積み重ねたような姿をしている(一次元のndarrayもある)
- axisという概念で行と列が区別される

最もシンプルにはリストに対して  
`np.array()`を適用して生成される



```
list_A = [0, 1, 2]  
array_A = np.array(list_A)
```



```
[0 1 2]  
numpy.ndarray
```



# ndarray同士の四則演算ができる

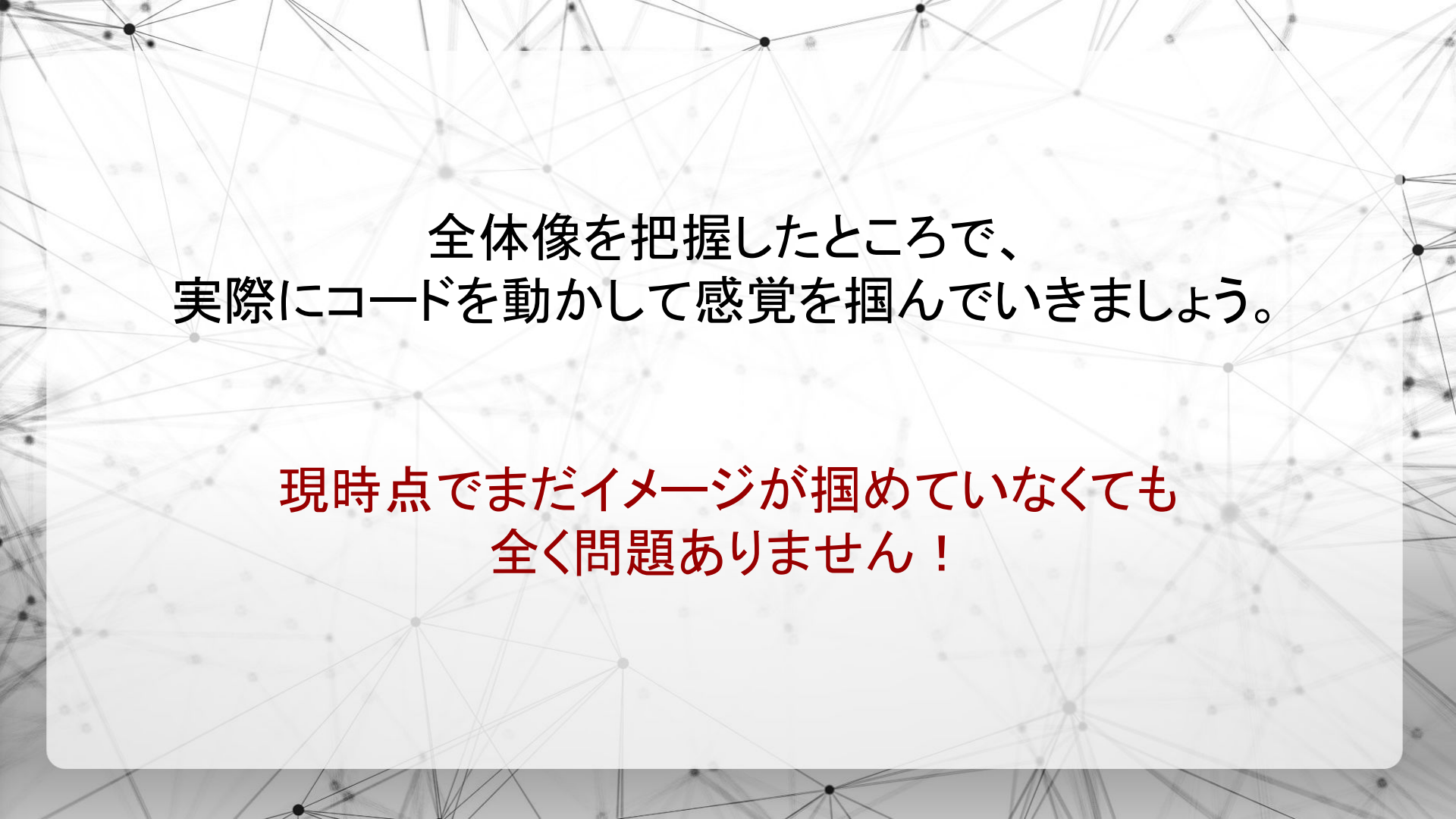


```
a = np.array([1, 2, 3, 4, 5])  
b = np.array([2, 2, 3, 6, 0])  
a + b
```

```
array([ 3,  4,  6, 10,  5])
```

# リスト型データとの対比

リスト	<code>numpy.ndarray</code>
異なる型の値を格納できる	同じ型の値しか格納できない
リスト同士の四則演算は煩雑	<code>ndarray</code> 同士の四則演算が容易
入れ子構造のリストは作れるが <code>axis</code> という概念がないため疑似的にしか多次元配列を扱うことができない	多次元配列を扱える

The background of the slide features a complex network of thin, light gray lines connecting numerous small, dark gray dots. These dots are scattered across the entire frame, creating a web-like or molecular structure that suggests connectivity and data flow. The overall aesthetic is technical and modern.

全体像を把握したところで、  
実際にコードを動かして感覚を掴んでいきましょう。

現時点でまだイメージが掴めていなくても  
全く問題ありません！



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

```
b = np.array([2, 2, 3, 6, 0])
```

```
a * b
```

```
↳ array([ 2,  4,  9, 24,  0])
```





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

```
a * b
```

```
↳ array([ 2,  4,  9, 24,  0])
```

The diagram illustrates the element-wise multiplication of two NumPy arrays, `a` and `b`. The first array `a` contains the values `[1, 2, 3, 4, 5]`, and the second array `b` contains the values `[2, 2, 3, 6, 0]`. The operation `a * b` is performed, resulting in a new array `array([ 2, 4, 9, 24, 0])`. Red boxes are drawn around the elements of `a` and `b` that are multiplied to produce each element of the result. Red arrows point from these boxes to the corresponding elements in the resulting array: `1 * 2 = 2`, `2 * 2 = 4`, `3 * 3 = 9`, `4 * 6 = 24`, and `5 * 0 = 0`.



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

$a * b$

→ `array([ 2, 4, 9, 24, 0])`

# ユニバーサルな計算が便利な例

	世帯所得	築年数	部屋数	ベッドルーム数	人口	占有率	緯度	経度
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25

各カラムごとの全レコードの中央値や標準偏差などを計算したい！そういうときに便利。



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

```
b = np.array([ 2 , 2 , 3 , 6 , 0 ])
```

```
a == b
```

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





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

```
a == b
```

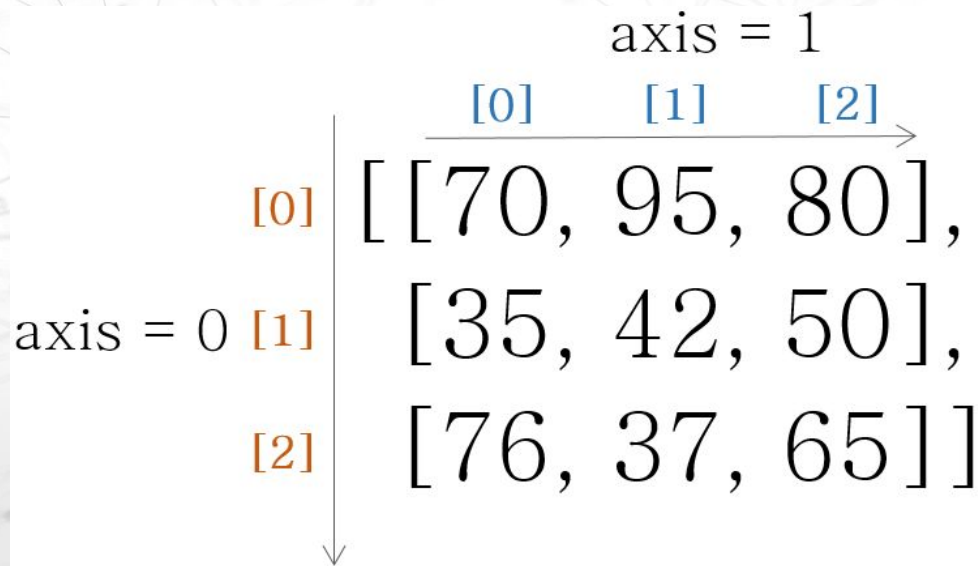
→ `array([False, True, True, False, False])`

# numpy.ndarrayとはn次元配列データ

axis = 1

	[0]	[1]	[2]
[0]	70	95	80
[1]	35	42	50
[2]	76	37	65

axis = 0



- ndarrayはリスト型を積み重ねたような姿をしている(一次元のndarrayもある)
- axisという概念で行と列が区別される

# スライシングのstart, end, step

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

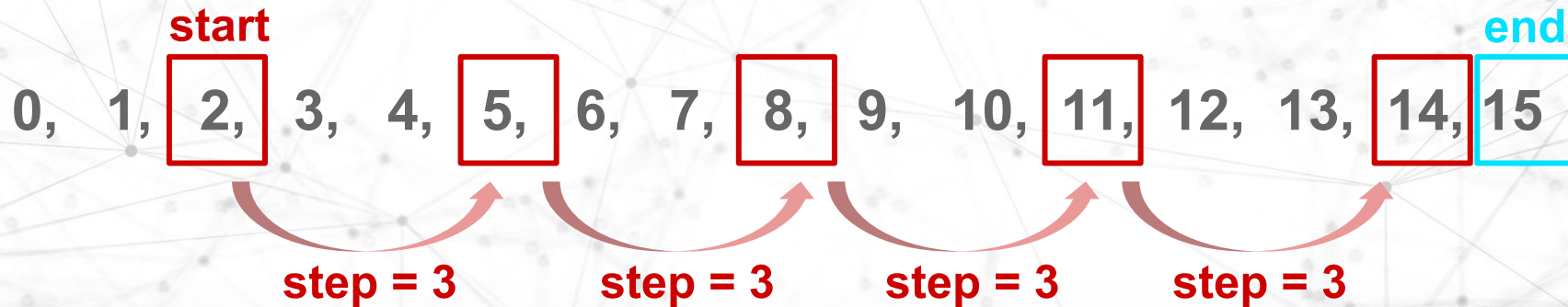
✓  
0  
秒



```
a = list(range(30)) # 0~29を要素にもつList  
print(a[2: 15: 3])
```

```
☐➡ [2, 5, 8, 11, 14]
```

# スライシングのstart, end, step



✓  
0  
秒



```
a = list(range(30)) # 0~29を要素にもつList  
print(a[2: 15: 3])
```

```
☞ [2, 5, 8, 11, 14]
```



# ネスト構造のリスト

[<sup>[0]</sup> 1, <sup>[1]</sup> 2, <sup>[2]</sup> 3]

整数型の値を要素に持つリストに対してnp.array()を適用

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

[<sup>[0]</sup> [1, 2], <sup>[1]</sup> [3, 4], <sup>[2]</sup> [5, 6]]

リスト型の値を要素に持つリストに対してnp.array()を適用

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

# Numpyにおける行列の計算

行列a

1	2
3	4
5	6

行列b

1	1
2	3
5	8

同じ位置の要素同士で四則演算  
＝ユニバーサル計算

1 <sup>1</sup>	2 <sup>1</sup>
3 <sup>2</sup>	4 <sup>3</sup>
5 <sup>5</sup>	6 <sup>8</sup>

行列aとbの積の場合

1	2
6	12
25	48

行列積とは異なる点に注意

# Numpyの2次元配列の構造

		axis = 1		
		[0]	[1]	[2]
axis = 0	[0]	[[70, 95, 80],		
	[1]	[35, 42, 50],		
	[2]	[76, 37, 65]]		

# 2次元配列としてのテーブルデータ

axis=1のindex → [0] [1] [2] [3] [4] [5] [6] [7]

	世帯所得	築年数	部屋数	ベッドルーム数	人口	占有率	緯度	経度
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25

各行: 一件一件のレコード(顧客や商品などの情報)

各列: 各レコードの特徴を記録したデータ(特徴量)



# 2次元配列としてのテーブルデータ

axis=1 → 各特徴量

axis=0

各レコード

	世帯所得	築年数	部屋数	ベッドルーム数	人口	占有率	緯度	経度
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25

各行: 一件一件のレコード(顧客や商品などの情報)

各列: 各レコードの特徴を記録したデータ(特徴量)



## 2次元配列への集約関数の適用

axis=0

$\begin{bmatrix} [88, 78, 76], \\ [98, 88, 100], \\ [64, 78, 77], \\ [89, 67, 78] \end{bmatrix}$

max  $[98, 88, 100]$

```
[[ 88  78  76]
 [ 98  88 100]
 [ 64  78  77]
 [ 89  67  78]]
```

```
print("①: ", np.max(a, axis=0))
```

```
①: [ 98  88 100]
```

axis=0を引数としたnp.max()の場合、スコープを縦軸にとり、全ての列に対して、「その列の中での最大値」を返す。

## 2次元配列への集約関数の適用

axis=1 → max

[ [ 88, 78, 76 ] ,	→	88
[ [ 98, 88, 100 ] ,	→	100
[ [ 64, 78, 77 ] ,	→	78
[ [ 89, 67, 78 ] ]	→	89

```
[[ 88 78 76]
 [ 98 88 100]
 [ 64 78 77]
 [ 89 67 78]]
```

```
print("②: ", np.max(a, axis=1))
```

```
②: [ 88 100 78 89]
```

axis=0を引数としたnp.max()の場合、スコープを縦軸にとり、全ての列に対して、「その列の中の最大値」を返す。

# 「行列a」に対するインデックス指定

行列a

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

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

# 「行列a」に対するインデックス指定

axis=1

[0] [1] [2] [3]

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

[0]

axis=0 [1]

[2]

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



# 「行列a」に対するインデックス指定

axis=1 [0] [1] [2] [3] →

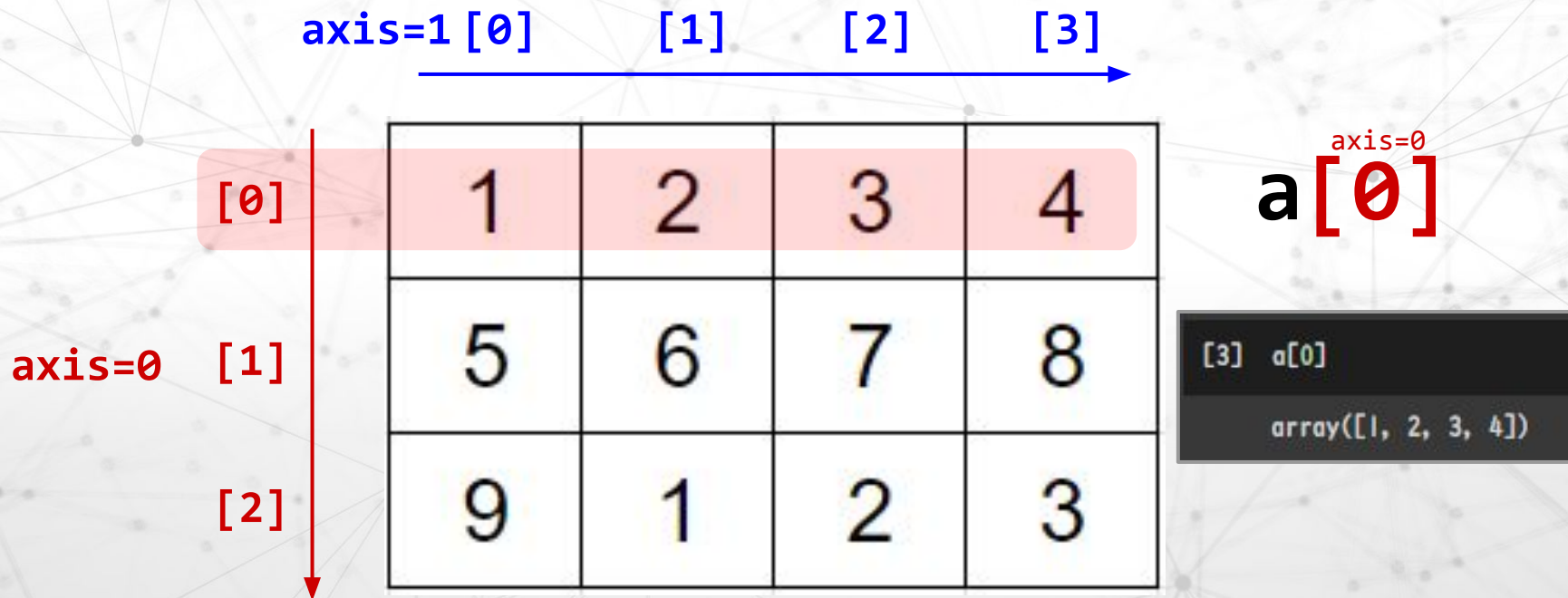
[0]	1	2	3	4
axis=0 [1]	5	6	7	8
[2]	9	1	2	3

axis=0 axis=1  
a[0][1]

```
[4] a[0][1]  
2
```



# axis=0に対する範囲指定



ndarrayの場合、カンマ区切りでも良い

$$\overset{\text{axis}=0}{a}[\overset{\text{axis}=1}{0}, \overset{\text{axis}=1}{1}] = \overset{\text{axis}=0}{a}[\overset{\text{axis}=1}{0}][\overset{\text{axis}=1}{1}]$$

axis=0

[0]

[1]

[2]

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

[0] [1] [2] [3]

axis=1

```
[4] a[0][1]
0
2
```

```
[12] a[0, 1]
0
2
```

```
✓ 0 秒 # axis 0の0番目と2番目を取ってくる  
a[[0, 2]] # a[0, 2]とは違う  
array([[1, 2, 3, 4],  
       [9, 1, 2, 3]])
```

axis=0

[0]

[1]

[2]

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

[0]

[1]

[2]

[3]

axis=1

【参考】  
要素を一つだけ指定する場合

axis=0  
**a[0]**

axis=0  
**a[[0, 2]]**

【POINT】  
コレとの違いを押さえましょう

axis=0 axis=1  
**a[0, 2]**

axis=1

	[0]	[1]	[2]	[3]
axis=0				
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	1	2	3

a<sup>axis=0</sup><sup>axis=1</sup>[[0,1],[1,2]]

```
[17] a[[0, 1], [1, 2]]
      array([2, 7])
```

【参考】  
axis=1を指定しない場合

<sup>axis=0</sup>  
a[[0,1]]

【参考】  
要素を一つだけ指定する場合

<sup>axis=0</sup> <sup>axis=1</sup>  
a[0,2]

# 一つの軸から複数の要素を指定(1)

```
axis=0    axis=1  
[12] a[2, [1, 3]]  
array([1, 3])
```

axis=1 [0] [1] [2] [3]

axis=0 [0] [1] [2]

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



# 一つの軸から複数の要素を指定(2)

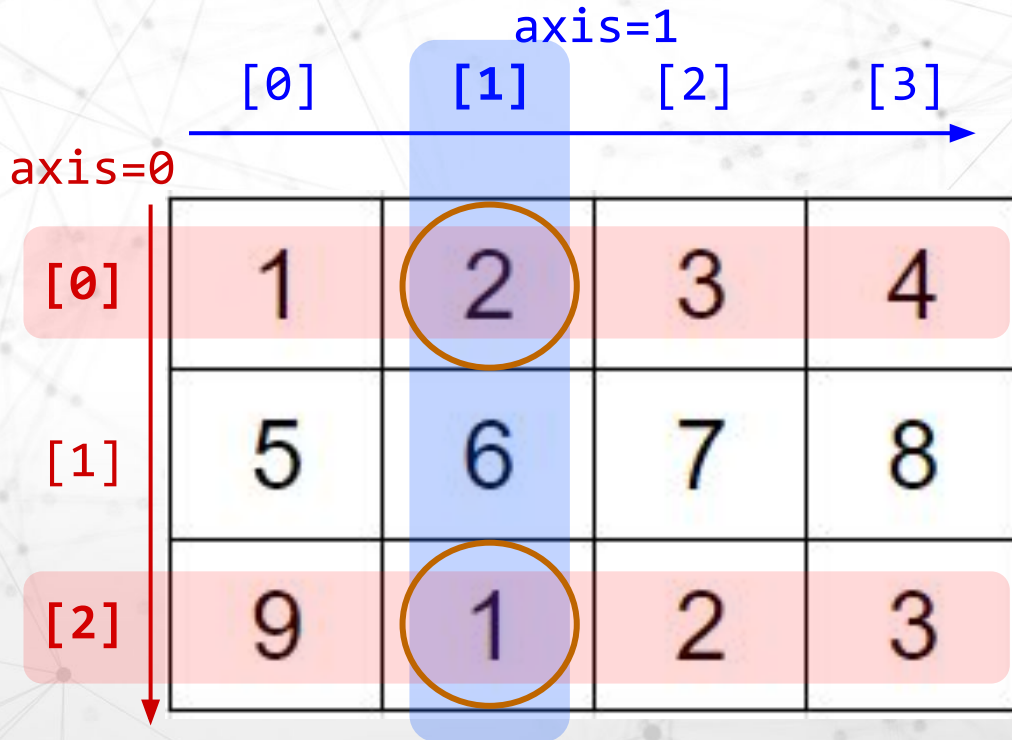
axis=0に指定するリストや配列は  
axis=0と同じ縦向きにする

```
a[[0], [2], 1]  
axis=0 axis=1
```

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

```
axis=0 axis=1  
a[[0, 2], 1]
```

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



# 複数の軸から複数の要素を指定

axis=0に指定するリストや配列は axis=0と同じ縦向きにする

axis=0 axis=1  
`a[[0], [2]], [1, 3]]`

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

axis=0 axis=1  
~~`a[[0, 2], [1, 3]]`~~

~~`array([2, 3])`~~

これだと`a[0, 1]`, `a[2, 3]`  
を取ってきてしまう。

axis=1

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	1	2	3

axis=0に指定するリストや配列を  
縦向きにしないと、これになってしまう  
(飛び飛びの個別要素の抽出になる)

axis=0

[0]

[1]

[2]

1	2 <small>a[0,1]</small>	3	4
5	6	7	8
9	1	2	3 <small>a[2,3]</small>

[0] [1] [2] [3]

axis=1

axis=0 axis=1

a[[0,2],[1,3]]

axis=0 axis=1

~~a[[0,2],[1,3]]~~

~~array([2,3])~~

これだとa[0,1], a[2,3]  
を取ってきてしまう。

# 複数の軸から複数の要素を指定

axis=0に指定するリストや配列は axis=0と同じ縦向きにする

axis=0 axis=1  
`a[[0], [2]], [1, 3]]`

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

axis=0 axis=1  
~~`a[[0, 2], [1, 3]]`~~

~~`array([2, 3])`~~

これだと`a[0, 1]`, `a[2, 3]`  
を取ってきてしまう。

axis=1

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	1	2	3



**axis=1**の記載を省略すなわち**axis=0**だけ一部指定して  
**axis=1**の要素は全選択する場合のおさらい

axis=1を省略する場合

axis=0  
**a[0]**

axis=1を省略する場合

axis=0  
**a[[0,2]]**

axis=1

[0] [1] [2] [3]

[0]	1	2	3	4
axis=0 [1]	5	6	7	8
[2]	9	1	2	3

axis=1

[0] [1] [2] [3]

[0]	1	2	3	4
axis=0 [1]	5	6	7	8
[2]	9	1	2	3



では $\text{axis}=0$ の記載を省略すなわち $\text{axis}=1$ だけ一部指定して  
 $\text{axis}=0$ の要素は全選択する場合は??

$\text{axis}=1$ を省略する場合

$\text{a}[\text{0}]$

$\text{axis}=1$

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	1	2	3

$\text{axis}=1$ を省略する場合

$\text{a}[\text{[0,2]}]$

$\text{axis}=1$

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	1	2	3

**axis=0**のインデックス指定を省略することは  
できないので、**スライシング**を使う。

axis=1を省略する場合

axis=0  
**a[0]**

axis=1を省略する場合

axis=0  
**a[[0,2]]**

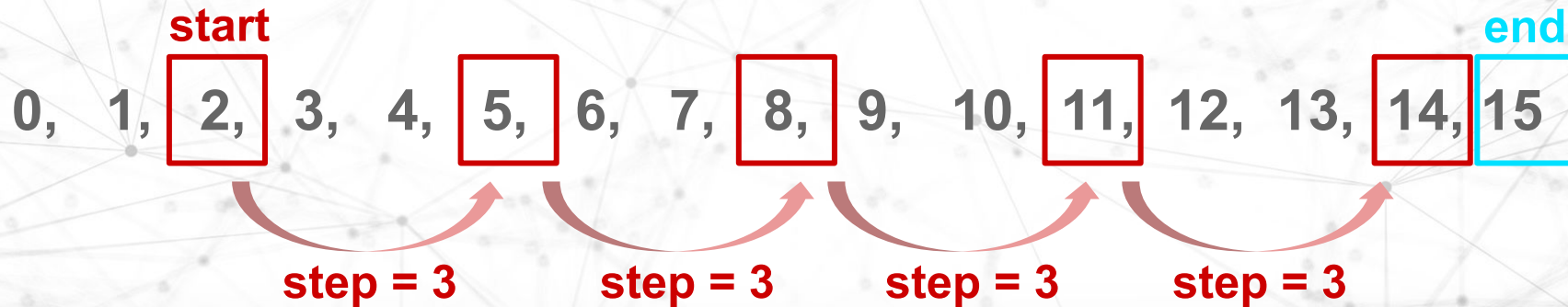
axis=0を省略する場合

axis=0 axis=1  
**a[:,0]**

axis=0を省略する場合

axis=0 axis=1  
**a[:, [0,2]]**

# スライシングのstart, end, step (おさらい)



✓  
0  
秒



```
a = list(range(30)) # 0~29を要素にもつList  
print(a[2: 15: 3])
```

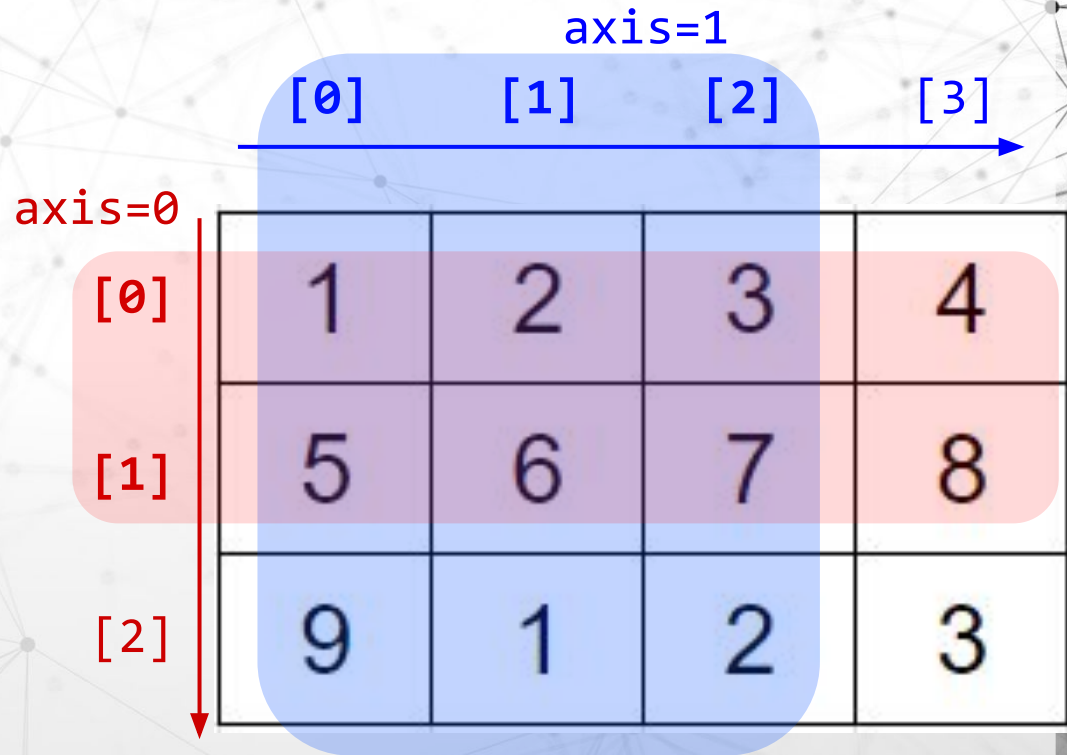
```
☞ [2, 5, 8, 11, 14]
```

# スライシングによる範囲指定

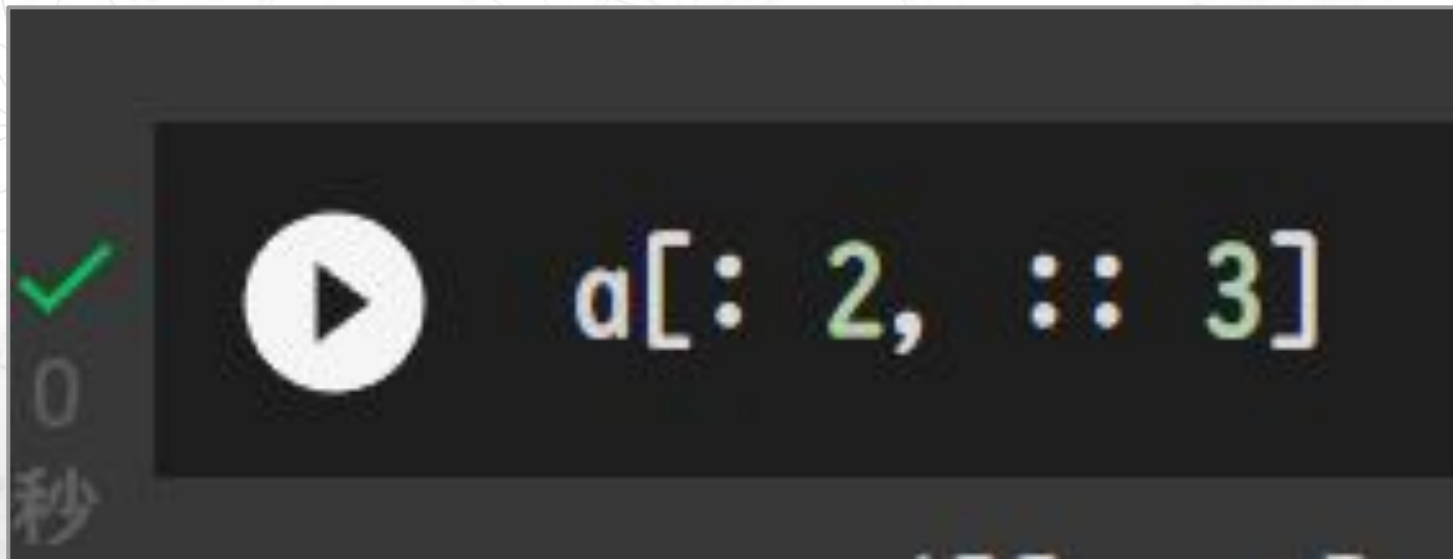
`a[0: 2, 0: 3]`

*axis=0*      *axis=1*

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



これは行列aのどの要素を指定したことになるでしょう？





# 答え



0  
秒

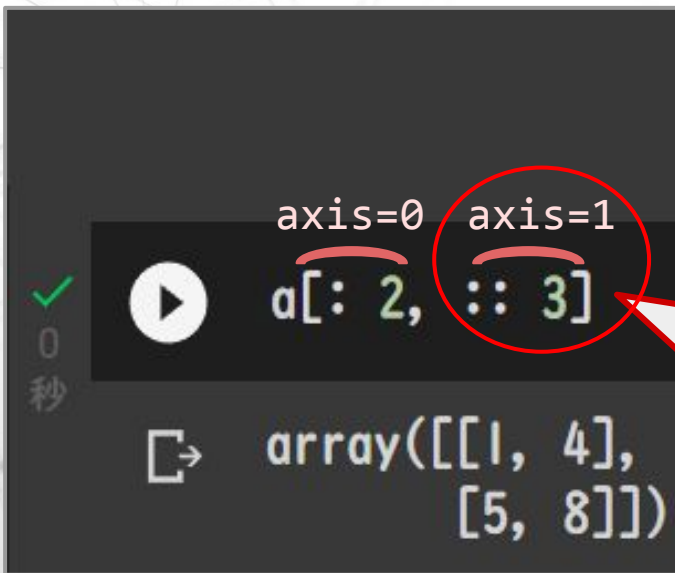


```
a[: 2, :: 3]
```



```
array([[1, 4],  
       [5, 8]])
```

# スライシングによる範囲指定



```
✓ 0 秒  
▶ a[:, 2, :: 3]  
↳ array([[1, 4],  
         [5, 8]])
```

axis=1のスライシングはこうなっています。

start

end

step

省略

:

省略

3

※axis=0はstepを省略している。

# スライシングによる範囲指定

✓  
0 秒



axis=0 axis=1  
`a[:, 2, :: 3]`

`array([[1, 4],  
[5, 8]])`

axis=1

[0] [1] [2] [3]

axis=0

[0]

[1]

[2]

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	1	2	3

行列a

2	2	3	6
0	6	7	9

`idx = (a%3 == 0)`

`a % 3 == 0`

ユニバーサル計算を適用

ブール値の行列「idx」

FALSE	FALSE	TRUE	TRUE
TRUE	TRUE	FALSE	TRUE

```

a = np.array([2, 2, 3, 6, 0, 6, 7, 9]).reshape(2, 4)
print("a: ", a)
# aのうち3の倍数である値を抜き出す
idx = a%3 == 0
print("ブールインデックス: ¥n", idx)
print(type(idx))
print(a[idx])

a: [[2 2 3 6]
     [0 6 7 9]]
ブールインデックス:
[[False False True True]
 [ True True False True]]
<class 'numpy.ndarray'>
[3 6 0 6 9]
    
```

↑今ノートブックでみていたコード

FALSE	FALSE	TRUE	TRUE
TRUE	TRUE	FALSE	TRUE



出力結果は一次元配列

3	6	0	6	9
---	---	---	---	---