

Python活用基礎研修: 入門編2

NumPyを使ったPythonプログラミング

Python活用基礎研修について

研修の目標

- Pythonを通してプログラミングの基礎を学ぶ.
- 実際に手を動かして簡単なデータ解析およびAI（機械学習）を体験する.

入門編

- 入門編1: Pythonプログラミングの基本的なルールを学ぶ.
- 入門編2: NumPyを用いたプログラミングを学ぶ.

応用編: 各種モジュール（=便利なツール）の使い方を学ぶ

- 応用編1（データ解析）: pandas, Matplotlib
- 応用編2（AI, 機械学習）: scikit-learn

お詫びと訂正

入門編1のスライドに研修環境は**Jupyter Notebook**を使用すると記述しておりましたが、第0回資料に従って、正しくは**JupyterLab**を使用いたします。
操作方法等の説明において混乱を招いてしまい申し訳ございません。

研修の進行速度および理解度把握のためのご協力をお願い

本日の研修では、研修の進行速度が適切かを把握するために、Zoomの**リアクション機能**を用いたレスポンスをお願いすることがございます。研修中にこちらから進行速度が早すぎないか質問した際に、**リアクション機能**の絵文字で**進行速度**や**理解度**に対する感想をお願いいたします。

リアクションの例

- 👍, 👉 : ちょうど良い または 理解できている
- 😬, 🤔 : すこし早い または 少し難しい
- 😓, 🥲 : もうすこしゆっくりが良い または 前の説明範囲で躓いている

前回の練習問題1の答え

リスト `a = [1, 3, 5, 7, 9]` の要素の合計値を `for` 文を用いて計算せよ.

`for` 文を用いることで反復処理を実行できる.

```
for 変数名 in リスト名:  
    処理1 # リストの要素を用いた処理などを記述  
    処理2  
    ...
```

```
a = [1, 3, 5, 7, 9]  
goukei = 0 # 合計値を管理するための変数goukeiを初期値0で準備.  
for i in a: # for文の記述ルールに従い, リストaの各要素を変数iで順次取得  
    goukei = goukei + i # リストaの ある要素iを変数goukeiに加算し値を更新  
print(goukei)
```

前回の練習問題2の答え

以下の成績表について

1. 山田の成績のリストを作り, 3科目の合計点を求めよ.

	名前	国語	数学	英語
0	山田	80	90	75
1	田中	50	95	67
2	鈴木	73	56	95

```
yamada_l = [80, 90, 75] # 表を元に山田の各科目の成績を管理するリストを作成
goukei = 0               # 合計値を管理するための変数goukeiを初期値0で準備.
for i in yamada_l:       # for文の記述ルールに従い, リストyamada_lの各要素を変数iで順次取得
    goukei = goukei + i # リストyamada_lの ある要素iを変数goukeiに加算し値を更新
print(goukei)
```

前回の練習問題2の答え

以下の成績表について

2. 数学の成績のリストを作り, 数学の平均点を求めよ.

	名前	国語	数学	英語
0	山田	80	90	75
1	田中	50	95	67
2	鈴木	73	56	95

```
math_l = [90, 95, 56] # 表を元に数学の各生徒の成績を管理するリストを作成
goukei = 0 # 合計値を管理するための変数goukeiを初期値0で準備.
for i in math_l: # for文の記述ルールに従い, リストmath_lの各要素を変数iで順次取得
    goukei = goukei + i # リストmath_lの ある要素iを変数goukeiに加算し値を更新
heikin = goukei / 3 # 平均値を算出するために合計値を3で割る
print(heikin)
```

前回の練習問題3の答え

一次方程式 $y = 2x + 1$ があるとき, $x_l = [-2, -1, 0, 1, 2]$ のリストを作成し, `for` 文を用いて, x の各値に対応する y の値を求めよ.

ヒント

y の値を管理するためのリスト `y = []` を用意する.

`y.append(a)` と記述することで, リスト `y` の末尾に要素 `a` を追加できる.

```
x_l = [-2, -1, 0, 1, 2] # 入力値のリストx_lを生成
y = []                  # 要素が1つも入っていない空のリストyを生成
for x in x_l:           # for文の記述ルールに従い, リストx_lの各要素を変数xで順次取得
    y.append(2*x + 1)    # 計算結果をappend関数を用いてリストyの末尾に追加
print(y)
```


入門編2: Numpyを使ったPythonプログラミング

0. NumPy

0.1. NumPyとは？

0.2. 復習: listについて

0.3. NumPyとlistの比較: 数値計算

0.4. 利用する準備

1. NumPyの基本

2. NumPyの関数

0.1 NumPyとは？

NumPyは数値計算を効率的に実行できるPythonのモジュール。

誕生の背景として, Pythonのみを用いた数値計算において, C言語やJavaと比較して, 実行時間が遅くなってしまうという欠点を抱えていたことが挙げられる。

この欠点を克服すべく1995年にTravis Oliphantらが開発したオープンソースソフトウェア。

主な特長

1. 強力な**N次元配列**を扱える点
2. 包括的な**数学関数**が提供されている点
3. C言語に基づく**高速な処理**が実現できる点

0.2. 復習: listについて

リストは, 複数の値や文字列をまとめて管理できるデータ構造.

具体的には, 下記のように数値や文字列などを `[]` の中に複数並べてることで定義する.

```
a = [1, 3, 5, 7, 9] # 数値のリスト
program = ['Python', 'C', 'Java', 'PHP'] # 文字列のリスト
```

インデックス

0	1	2	3	4
1	3	5	7	9
a[0]	a[1]	a[2]	a[3]	a[4]

リスト **a**

インデックス

0	1	2	3
'Python'	'C'	'Java'	'PHP'
program[0]	program[1]	program[2]	program[3]

リスト **program**

0.3. NumPyとlistの比較: 数値計算

1次元配列の各要素に数値**1**を加算する場合

listで実装すると...

```
a_list = [1, 3, 5, 7, 9]
new_list = []
for inner_a in a_list:
    new_list.append(inner_a + 1)
```

NumPyで実装すると...

```
import numpy as np
a_array = np.array([1, 3, 5, 7, 9])
a_array = a_array + 1
```

0.4. 利用する準備

NumPyはモジュールであるため利用するにはimportが必要.

```
import numpy as np
```

省略名の付与

プログラミングで多用するモジュールの名前は, 短い方が記述コストが低減する.

しかしモジュール名は必ずしも短いとは限らない.

そこで `as` という記述ルールを用いることでimportしたモジュール名を任意の名前へ変更できる. 上記の例では `numpy` を `np` へ変更している.

概要

0. NumPy

1. NumPyの基本

1.1. NumPy配列の定義

1.2. NumPy配列のプロパティ

1.3. 復習: 要素の参照

1.4. インデックスとスライス

1.5. 要素の置換

1.6. 練習問題

1.7. 数値計算 (スカラー値)

1.8. 数値計算 (行列演算)

1.9. 練習問題

2. NumPyの関数

1.1. NumPy配列の定義: 1次元配列の場合

NumPyでは, `ndarray` (N-dimensional array) というデータ構造を扱う.
listで扱っていたような**1次元配列**は `array` 関数を用いて以下のように定義する.

```
a_1d = np.array([1, 3, 5, 7, 9]) # 要素を5つ持つ1次元配列を生成  
print(a_1d)
```

1.1. NumPy配列の定義: 多次元配列の場合

多次元配列を扱いたい場合も同様の手順で `array` 関数を用いて定義できる.

以下は, 2x2の2次元配列および2x3x2の3次元配列を定義する例である.

```
# 要素を4個からなる2x2の2次元配列を生成
a_2d = np.array([[1, 2], [3, 4]])
print(a_2d)
```

```
# 要素を12個からなる2x3x2の3次元配列を生成
a_3d = np.array([[[1, 2], [3, 4], [5, 6]], [[7, 8], [9, 10], [11, 12]]])
print(a_3d)
```

listを用いた多次元配列

上記から分かるとおりlistでも多次元配列を定義可能.

ただしその利便性から特に数値計算においてはNumPyが利用される.

```
a_2d_list = [[1, 2], [3, 4]]
```


1.2. NumPy配列のプロパティ

生成した多次元配列の**プロパティ**（＝固有のデータ）の取得手段が実装されている。例えば、配列の次元数, 形状および要素数は, 生成された配列毎に異なる。

```
a_2d.ndim    # 2, 2次元配列
a_2d.shape   # (2, 2), 2x2の形状
a_2d.size    # 4, 要素数は4

a_3d.shape   # (2, 3, 2), 2x3x2の形状
```

プロパティの呼び出し

これまでモジュール名に`.`を付けることで**関数()**を呼び出していた。
それに対し上記の例では, 変数名に`.`を付け**プロパティ**を呼び出している。

1.3. 復習: 要素の参照

リストの各要素を参照したい場合, リスト名[要素の番号] と記述する.
また, この要素の番号のことを インデックス もしくは 添字 という.

インデックスは 0 から始まることに注意.

なお, インデックスは要素数を超えるものを指定できない.

a[0] # リストaの1つめの要素は, 添字0を指定することで格納された値1を参照可能
a[2] # リストaの3つめの要素は, 添字2を指定することで格納された値5を参照可能

インデックス				
0	1	2	3	4
1	3	5	7	9
a[0]	a[1]	a[2]	a[3]	a[4]

リスト a

1.4. インデックスとスライス: 1次元配列の場合

NumPyにおいて配列の各要素の参照は**インデックス**を用いる.

```
# a_1d = np.array([1, 3, 5, 7, 9])  
a_1d[0] # 1  
a_1d[4] # 9
```

また, より柔軟な参照方法として**スライス**を用いる.

具体的には, **:** とインデックスを用いることで以下のように参照範囲を指定できる.

特に, **終了位置のインデックスは参照範囲に含まれない**ことに注意.

```
# 配列名[開始位置:終了位置:増分]  
a_1d[2:4] # [5 7]      開始位置以上, 終了位置未満を取得  
a_1d[3:]  # [7 9]      開始位置以上を取得  
a_1d[:4]  # [1 3 5 7]   終了位置未満を取得  
a_1d[1::2] # [3 7]      開始位置以上から2個ずつ取得  
a_1d[:]   # [1 3 5 7 9] 全ての範囲を取得 (特殊な例)
```

1.4. インデックスとスライス: 多次元配列の場合

1次元配列と同様にインデックスとスライスを用いて値を参照できる.

```
# a_2d = np.array([[1, 2], [3, 4]])  
# [[2 3]   # 1行目  
#  [4 5]]  # 2行目  
  
a_2d[0]      # [1 2] 1次元目の添字0の要素である配列を参照 (行の参照)  
a_2d[:, 1]   # [2 4] 1次元目は全て参照し, 2次元目は添字1の要素を参照 (列の参照)  
a_2d[0, 1]   # 2      1次元目の添字0, 2次元目の添字1の要素である整数を参照 (下記と処理内容は実質同じ)  
a_2d[0][1]   # 2      1次元目の添字0の要素である配列 の 次元目の添字1の要素である整数を参照
```

リストにおけるスライス

スライスはリストでも使用することができる.

```
a_2d_list = [[1, 2, 3], [4, 5, 6]]  
a_2d_list[0, :2] # [1 2]
```

1.5. 要素の置換

インデックスとスライスと用いることで参照した値を**置換**できる.

```
b_1d = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])  
b_1d[1] = 1    # インデックスを指定して値を0から1へ変更  
b_1d[5:9] = 1  # スライスで指定した範囲の値全てを1へ変更
```

viewとcopy

Pythonでは, 変数に変数を代入することができる.

下記の例では, 2行目の時点では `a` も `b` も同じ値 `100` を持つ.

その後, 3行目で `a` の値が `50` へ変更されても `b` の値は `100` のまま.

```
a = 100
b = a # a=100, b=100
a = 50 # a=50, b=100
```

しかし, 変数が特定の型 (ここでは `ndarray` 型) の場合, その限りではない.

```
b_1d = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
c_1d = b_1d # 変数c_1dに変数b_1dを代入
b_1d[5:9] = 1 # 変数b_1dの配列を操作
print(b_1d)
print(c_1d) # b_1dの配列操作がc_1dにも反映されている
```

viewとcopy

NumPyの配列は, 配列の操作を行うときに `view` か `copy` のどちらかの状態で管理される. 例えば, 新しい変数に古い変数を代入するときは, 古い変数 (元の配列) の `view` を渡す.

- `view`: 元の配列とメモリを共有し, 値も全く同じ
- `copy`: 元の配列とメモリを共有せず, 操作時点の値を保持

```
b_1d = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
c_1d = b_1d      # 変数c_1dに変数b_1dを代入
b_1d[5:9] = 1    # 変数b_1dの配列を操作
print(b_1d)
print(c_1d)      # スライスによる操作はview. b_1dの配列操作がc_1dにも反映されている
b_1d = b_1d - 1  # 変数b_1dの配列を操作
print(b_1d)
print(c_1d)      # 加算による操作はcopy. b_1dの配列操作がc_1dにも反映されない
```

viewとcopy

deepcopy 関数を用いると確実に copy を取得できる.

```
import copy
b_1d = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
c_1d = copy.deepcopy(b_1d) # 変数c_1dに変数b_1dのcopyを代入
b_1d[5:9] = 1              # 変数b_1dの配列を操作
print(b_1d)
print(c_1d)                # b_1dの配列操作がc_1dにも反映されない
```


1.6 練習問題

3x3の配列が代入された変数 `c_3d` について

```
c_3d = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
```

以下の配列になるように変数 `c_3d` を操作せよ.

```
[[0 0 0]
 [0 1 0]
 [0 0 0]]
```

1.7. 数値計算（スカラー値）

NumPyには、ブロードキャストという演算を補助する機能が備わっている。
まず、配列と**スカラー値**（ベクトル等ではない単一の値）の計算について確認する。
下記の例では、2x2の配列にスカラー値**1**の加算などを実行している。

```
# a_2d = np.array([[1, 2], [3, 4]])  
a_2d + 1          # 2x2の全ての要素に1を加算  
a_2d[:, 1] + 1    # インデックスとスライスで参照した列の要素のみに1を加算  
a_2d * 2          # 2x2の全ての要素に2を乗算
```

上記のようにスカラー値と配列間で四則演算を実行した場合、
配列の全ての要素に対して計算が行われる。
これによりPython上で **for** 文などを繰り返し処理を用いない計算が可能となる。

1.8. 数値計算（行列演算）：同じ形状の場合

次に, 同じ形状の配列同士の計算について確認する.

```
# a_2d = np.array([[1, 2], [3, 4]])  
b_2d = np.array([[1, 2], [4, 8]])  
a_2d + b_2d  
# 対応する要素同士が加算される  
# [[ 2  4]  
#   [ 7 12]]
```

1.8. 数値計算（行列演算）：異なる形状の場合

最後に、異なる形状の配列同士の計算について確認する.

```
# a_2d = np.array([[1, 2], [3, 4]])
# a_3d = np.array([[[1, 2], [3, 4], [5, 6]], [[7, 8], [9, 10], [11, 12]]])
# 配列a_2dの形状は2x2, 配列a_3dの形状は2x3x2
a_2d + a_3d # 配列間の形状が異なるため計算が実行できずエラーが出力される
```

下記の例は**ブロードキャスト**により、2x2の配列と2x2x2の配列の演算が可能

```
b_3d = np.array([[[1, 2], [3, 4]], [[7, 8], [9, 10]]]) # 配列b_3dの形状は2x2x2
a_2d + b_3d
# a_2dの値が, b_3dの各1次元目に加算されている
# [[ 2  4]  =  [[ 1  2]  +  [[ 1  2]
#  [ 6  8]]  =  [ 3  4]]  +  [ 3  4]]
#
# [[ 8 10]  =  [[ 1  2]  +  [[ 7  8]
#  [12 14]]] =  [ 3  4]]  +  [ 9 10]]
```

1.8. 数値計算（行列演算）：ブロードキャスト

ブロードキャストは、異なる形状の配列間演算を可能にするための**形状の自動変換**のこと。
ただし**特定のルールに従って自動変換**されるため、
形状の変換によって**配列間の計算が必ず実現するわけではない**ことに注意。

形状の自動変換手順

1. 配列間の次元数を揃える

例えば、4x2の2次元配列 `org_A_2d` と2x4x1の3次元配列 `org_B_3d` で計算する場合、
配列Aの次元数を2次元 `org_A_2d` から3次元 `rule1_A_3d` へ自動変換する。

その際、配列の先頭に次元数を追加して、4x2から1x4x2の形状へ自動変換される。

```
org_A_2d = np.array([[0,1],[2,3],[4,5],[6,7]])           # 形状は4x2
org_B_3d = np.array([[[0],[1],[2],[3]],[[4],[5],[6],[7]]) # 形状は2x4x1
rule1_A_3d = np.array([[[0,1],[2,3],[4,5],[6,7]]])        # 形状を4x2から1x4x2へ変換
```

1.8. 数値計算（行列演算）：ブロードキャスト

形状の自動変換手順

2. 次元の要素数1が場合, もう一方の次元数と同じ数値になるよう値を繰り返す

ルール1により配列 `rule1_A_3d` と配列 `org_B_3d` は同じ次元数になった.

行列として演算するためには, それぞれの次元数の長さを同じにする必要がある.

`rule1_A_3d` は1つめの次元を2へ, `org_B_3d` は3つめの次元を2なるよう値を複製.

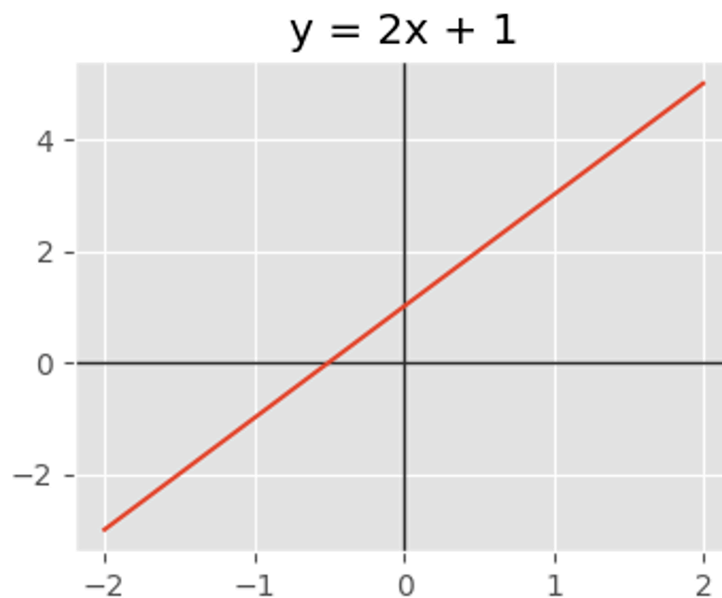
```
rule1_A_3d = np.array([[[0,1],[2,3],[4,5],[6,7]]])      # 形状を1x4x2
org_B_3d = np.array([[[0],[1],[2],[3]],[[4],[5],[6],[7]]]) # 形状は2x4x1

rule2_A_3d = np.array([[[0,1],[2,3],[4,5],[6,7]],[[0,1],[2,3],[4,5],[6,7]]]) # 形状を2x4x2
rule2_B_3d = np.array([[[0,0],[1,1],[2,2],[3,3]],[[4,4],[5,5],[6,6],[7,7]]]) # 形状は2x4x2
```

```
org_A_2d + org_B_3d      # ブロードキャストにより, 配列の形状が一致したため演算が可能
rule2_A_3d + rule2_B_3d  # 人カブロードキャストにより, 配列の形状が一致したため演算が可能
```

1.9. 練習問題

一次方程式 $y = 2x + 1$ について, $x_1d = [-2, -1, 0, 1, 2]$ の配列をNumPyで作成し, x_1d の各値に対応する y の値を求めよ.



概要

0. NumPy

1. NumPyの基本

2. **NumPy**の関数

2.1. 配列の初期化

2.2. 形状の変更

2.3. データ解析で用いる関数

2.4. 数学関数

2.5. 行列演算

2.6. 練習問題

はじめに: 関数の引数

Pythonの関数の引数（=入力）には、実行のために**必須の引数**と**省略可能な引数**がある。
例えば `print` 関数の引数は5つあり、それぞれ `objects`, `sep`, `end`, `file`, `flush` である。
このうち本研修では `objects` のみを引数として `print` 関数を実行していた。
つまり、`objects` が必須の引数、`sep`, `end`, `file`, `flush` は省略可能な引数である。

```
print(*objects, sep=' ', end='\n', file=None, flush=False)
```

省略可能な引数は、値が予め与えられているため引数を与えなくても関数を実行できる。
例えば引数 `end` は、`print` 関数を実行した時の末尾に入れる文字を指定できる。
また引数 `end` は、改行を意味する特別な記号 `\n` が予め与えられている。

```
print('引数endを指定しない場合は、文末は改行される。')  
print('引数endとして空の文字列を指定すると、文末は改行されない.', end='')  
print('引数endとして任意の文字列を指定すると、文末にその文字列が追加される.', end='[文末]\n')
```

2.1. 配列の初期化

NumPyには、配列の中身を特定のルールで生成する関数が実装されている。
特に、配列の**初期化**に用いられることが多い。

(**初期化**とは、変数の宣言と同時に何かしらの値を格納すること.)

```
a_2d = np.array([[1, 2], [3, 4]]) # 変数a_2dを初期化している
print(a_2d)
```

具体例として、全ての要素が0の配列を作成する `zeros` 関数などがある。

```
a0_2d_old = np.array([[0, 0], [0, 0]]) # これまでの初期化
a0_2d_new = np.zeros((2, 2), dtype=int) # 関数zerosを用いた初期化
print(a0_2d_old)
print(a0_2d_new)
```

2.1. 配列の初期化: 同じ値で埋める

初期化に用いられる関数

- zeros: 全ての要素が0の配列を生成
- ones: 全ての要素が1の配列を生成
- full: 全ての要素が引数で指定した任意の値で配列を生成

```
# shape(必須): 生成する配列の形状. 例えば2x2なら (2,2) と書く
# dtype(任意): 生成する配列の要素の型. 何も指定しない場合, 値は実数 (float型) になる.
# fill_value (必須): 関数fullのみ引数. 初期化に使う値を指定.
np.zeros(shape, dtype)
np.ones(shape, dtype)
np.full(shape, fill_value, dtype)
```

```
np.zeros((2, 2))           # 形状が2x2, 全ての値が実数0.0の配列. (dtypeが未指定のため実数)
np.ones((2, 2, 1), int)    # 形状が2x2x1, 全ての値が整数1の配列.
np.full((2, 3), 99)        # 形状が2x3, 全ての値が整数99の配列. (dtypeはfill_valueに基づいて決定)
```

2.1. 配列の初期化: 乱数で埋める

初期化に用いられる関数

- `rand`: 0.0以上1.0未満の連続一様分布に従う実数の乱数の配列を生成
- `randint`: 指定した範囲の離散一様分布に従う整数の乱数の配列を生成
- `randn`: 標準正規分布に従う実数の乱数の配列を生成

ここでは関数 `rand` の引数を確認する.

```
# d0(必須): 生成する配列の1つめの次元の長さ  
# d1, d2, ..., dn (任意): 生成する配列の各次元の長さ  
np.random.rand(d0, d1, d2, ..., dn)
```

<code>np.random.rand(2, 2)</code>	# 形状が2x2, 0.0以上1.0未満の連続一様分布に従う実数の乱数の配列.
<code>np.random.randint(1, 6, 10)</code>	# 形状が10, 1から6までの離散一様分布に従う整数の乱数の配列.
<code>np.random.randn()</code>	# 標準正規分布に従う実数の乱数を1つ返す (配列ではないことに注意).
<code>np.random.randn(2, 5)</code>	# 形状が2x5, 全ての値が標準正規分布に従う実数の乱数の配列.

2.1. 配列の初期化: 等差数列で埋める

他に指定した条件に従う等差数列を配列として生成する `arange` 関数が実装されている。

```
# start (任意): 数列の開始値. 指定しない場合, 値は0. 引数の値は生成する等差数列に含まる  
# stop (必須): 数列の終了値. ただし引数の値は生成する等差数列に含まれない  
# step (任意): 数列の間隔. 指定しない場合, 値は1  
# dtype (任意): 生成する配列の要素の型. 何も指定しない場合, 他の引数から自動的に決定される  
np.arange(start, stop, step, dtype)
```

```
np.arange(0, 10, 2) # 0から始まり9まで間隔2の等差数列  
np.arange(8)        # 0から始まり7まで間隔1の等差数列  
np.arange(2, 8)     # 2から始まり7まで間隔1の等差数列  
np.arange(0, 1, 0.1) # 0から始まり1まで間隔0.1の等差数列
```

2.2. 形状の変更

NumPyには, 生成した配列の形状を任意に変更する関数が実装されている.

`reshape` 関数は配列の形状を任意に変更できる. ただし変更前後の要素数は同じ.
また `reshape` 関数は, 配列の変数自身にも付属しており `a.reshape()` と書くことも可能.

```
# a (必須) : 変更したい配列の変数.  
# shape (必須) : 変更したい形状.  
np.reshape(a, shape)  
a.reshape(shape) # 上記の関数呼び出しと同じ結果になる
```

```
b0_1d = np.zeros(9)           # 形状が 9, 全ての値が実数0.0の配列  
np.reshape(b0_1d, (3, 3))    # 形状を 9, から 3x3 へ変更. NumPyの関数を使用  
b0_1d.reshape((3, 3))       # 形状を 9, から 3x3 へ変更. 変数b0_1dが持つ関数を使用
```

2.3. データ解析で用いる関数

NumPyには計算を伴うさまざまな関数が実装されている。
ここでは関数 `sum` の引数を確認する。

```
# a (必須) : 合計値を算出する対象となる配列
# axis (任意) : どの次元方向に計算処理を行うかを指定。
#               指定しない場合, 全ての要素について計算。
np.sum(a, axis=None)
```

```
# a_1d = np.array([1, 3, 5, 7, 9])
# a_2d = np.array([[1, 2], [3, 4]])
np.sum(a_2d)           # 10, 合計値を算出
np.sum(a_2d, axis=0)   # [4 6], 配列a_2dの1つめの次元方向にのみ合計値を算出
np.sum(a_2d, axis=1)   # [3 7], 配列a_2dの2つめの次元方向にのみ合計値を算出
np.mean(a_1d)          # 5.0, 平均値を算出
np.std(a_1d)           # 2.828427, 標準偏差を算出
np.var(a_1d)           # 8.0, 分散を算出
```

2.4. 数学関数

NumPyにはさまざまな数学関数が実装されている.

```
# 数学関数
np.sin() # 三角関数
np.sqrt() # 平方根
np.exp() # 指数関数
```

```
# a_1d = np.array([1, 3, 5, 7, 9])
np.sqrt(a_1d) # 各要素の平方根を適用: [1. 1.73205081 2.23606798 2.64575131 3.]
np.exp(a_1d) # 各要素の指数関数を適用: [2.71828183 20.0855369 148.413159 1096.63316 8103.08393]
```


2.5. 行列演算

NumPyには行列演算に特化した関数が実装されている.

```
np.dot()           # 行列積の計算  
np.linalg.det()    # 行列式の計算  
np.linalg.inv()    # 逆行列の計算  
np.linalg.eig()    # 固有値と固有ベクトルの計算
```

```
A_3d = np.array([[1, 1, -1], [-2, 0, 1], [0, 2, 1]]) # 3x3の配列A_3dを生成  
A_inv = np.linalg.inv(A_2d)                          # A_2dの逆行列を算出  
  
np.dot(A_3d, A_inv) # A_3dと逆行列の行列積を算出  
np.dot(A_inv, A_3d) # 上記と同様
```

2.6 練習問題: その1

以下の成績表について,

1. 3x3の配列をNumPyで生成せよ.
2. `sum` 関数を用いて, 山田の3科目の合計点を求めよ.
3. `mean` 関数を用いて, 数学の平均点を求めよ.

	名前	国語	数学	英語
0	山田	80	90	75
1	田中	50	95	67
2	鈴木	73	56	95

2.6 練習問題: その2

`array` 関数を用いずに以下の配列をそれぞれ生成せよ.

1. 値が3から始まり19で終わる, 奇数のみで構成された1次元配列

```
[3 5 7 9 11 13 15 17 19]
```

2. 値が0から始まり8で終わる, 3x3の配列

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

2.6 練習問題: その3

1. `array` 関数を用いずに以下の配列を変数 `f_2d` として生成せよ.

```
[[0. 0. 0. 0.]  
 [0. 1. 1. 0.]  
 [0. 1. 1. 0.]  
 [0. 0. 0. 0.]
```

2. 以下の配列 `g_2d` は配列 `f_2d` と同じ形状 4x4 の配列である.

配列 `g_2d` について, 配列 `f_2d` で値が1の範囲のみ合計値を求めよ.

```
g_2d = np.array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]])  
# [[ 0  1  2  3]  
#   [ 4  5  6  7]  
#   [ 8  9 10 11]  
#   [12 13 14 15]]
```