第7章 | NumPy

Author: sharo

7.1 Numpyの概観

7.1.1 NumPyとは

NumPyとは、Pythonでベクトルや行列計算を高速で行うのに特化した基盤となるライブラリです。

7.1.2 NumPyの高速な処理の体験

P174-175のコードを実行してみましょう。

7.2 NumPy1次元配列

7.2.1 import

NumPyをimportする際は

```
import numpy
```

と表記します。

```
import numpy as np
```

と表記することでパッケージ名を変更することができます。

7.2.2 1次元配列

NumPyには配列を高速に扱うための**ndarray**クラスが用意されています。**ndarray**を生成する方法の1つは、**NumPy**の `np.array()` 関数を用いる方法です。`np.array(リスト)` と表記し、リストを与えることで生成します。

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

また、`np.arange()` 関数を用いる方法があり、`np.arange(X)` と表記し、等間隔に増減させた値の要素をX個生成してくれます。

```
np.arange(4) # 出力結果 [0 1 2 3]
```

7.2.2ex 多次元配列

- 1次元(ベクトル)の**ndarray**クラス

```
array_1d = np.array([1, 2, 3, 4, 5, 6, 7, 8])
```

- 2次元(行列)の**ndarray**クラス

```
array_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
```

- 3次元(テンソル)の**ndarray**クラス

```
array_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

7.2.3 1次元配列の計算 - 1

リストでは、要素ごとの計算を行うためにはループを書いて要素を1つずつ取り出して計算を行う必要がありましたが、**ndarray**ではループで書く必要はありません。**ndarray**同士の算術演算では、同じ位置にある要素同士で計算されます。

7.2.3 1次元配列の計算 - 2

```
# NumPyを使わない場合
storages = [1, 2, 3, 4]
new_storages = []
for n in storages:
    n += n
    new_storages.append(n)
print(new_storages) # 出力結果: [2, 4, 6, 8]
```

7.2.3 1次元配列の計算 - 3

```
# NumPyを使う場合
import numpy as np
storages = np.array([1, 2, 3, 4])
storages += storages
print(storages) # 出力結果: [2 4 6 8]
```

7.2.4 インデックス参照とスライス

リスト型と同様にインデックス参照やスライスを行うことができます。

```
# スライス
arr = np.arange(10)
print(arr) # 出力結果: [0 1 2 3 4 5 6 7 8 9]
arr[0:3] = 1
print(arr) # 出力結果: [1 1 1 3 4 5 6 7 8 9]
```

7.2.5 ndarrayの注意点

ndarrayはリストと同じように代入先の変数の値を変更すると元の**ndarray**配列の値も変更されます。そのため、**ndarray**をコピーして2つの別々の変数にしたい場合は、`copy()`メソッドを使用します。

```
import numpy as np
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = arr1.copy()
arr2[0] = 100
print(arr1) # 出力結果: [1 2 3 4 5]
```

7.2.6 viewとcopy - 1

リストと**ndarray**の相違点としては、**ndarray**のスライスは配列のコピーではなく**view**であることです。**view**とは、*もとの配列と同じデータを指していること*を指します。

ndarrayのスライスの変更は、オリジナルの**ndarray**を変更するということになります。前節で確認した通り、スライスをコピーとして扱いたい場合には`arr[:].copy()`とします。

7.2.6 viewとcopy - 2

```
import numpy as np
# リストのスライス
arr_list = [x for x in range(10)]
print("arr_list: ", arr_list)
arr_list_copy = arr_list[:]
arr_list_copy[0] = 100
print("arr_list: ", arr_list)

# ndarrayのスライス
arr_np = np.arange(10)
print("arr_np: ", arr_np)
arr_np_view = arr_np[:]
arr_np_view[0] = 100
print("arr_np: ", arr_np)
```

7.2.7 ブールインデックス参照

```
arr = np.array([2, 4, 6, 7])
print(arr[np.array([True, True, True, False])]) # 出力結果: [2 4 6]
```

```
arr = np.array([2, 4, 6, 7])
print(arr[arr % 3 == 1]) # 出力結果: [4 7]
```

7.2.8 ユニバーサル関数

ユニバーサル関数とは**ndarray**配列の各要素に対して演算した結果を返す関数のことです。

```
import numpy as np
arr = np.array([4, -9, 16, -4, 20])
print(arr) # 計算結果: [ 4 -9 16 -4 20]
arr_abs = np.abs(arr)
print(arr_abs) # 計算結果: [ 4  9 16  4 20]
```

7.2.9 集合関数

集合関数とは**数学の集合演算を行う関数**のことです。**1次元配列のみ**を対象としています。

```
import numpy as np
arr1 = [2, 5, 7, 9, 5, 2]
arr2 = [2, 5, 8, 3 ,1]
# np.unique()関数を用いて重複をなくした配列を変数new_arr1に代入
new_arr1 = np.unique(arr1)
print(new_arr1) # 出力結果: [2 5 7 9]
# 変数new_arr1と変数arr2の和集合を出力
print(np.union1d(new_arr1, arr2)) # 出力結果: [1 2 3 5 7 8 9]
# 変数new_arr1と変数arr2の積集合を出力
print(np.intersect1d(new_arr1, arr2)) # 出力結果: [2 5]
# 変数new_arr1から変数arr2を引いた差集合を出力
print(np.setdiff1d(new_arr1, arr2)) # 出力結果: [7 9]
```

7.2.10 乱数

NumPyでは**np.random**モジュールで乱数を生成することができます。

```
import numpy as np # numpyをnpとしてimport
from numpy.random import randint # numpy.random.randintをrandintとしてimport
# 変数arr1に、各要素が0以上10以下の整数の行列(5 × 2)を代入
arr1 = randint(0, 11, (5, 2))
print(arr1) # 出力結果(例): [[ 3  8] [ 0  8] [ 4  2] [ 7 10] [ 2  7]]
# 変数arr2に0~1までの一様乱数を3つ代入
arr2 = np.random.rand(3)
print(arr2) # 出力結果(例): [0.04966003 0.96471596 0.01744918]
```

7.3 NumPy2次元配列

7.3.1 2次元配列

```
import numpy as np
# 変数arrに2次元配列を代入
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
# 変数arrの行列の各次元毎の要素数を出力
print(arr.shape) # 出力結果: (2, 4)
# 変数arrを4行2列の行列に変換
print(arr.reshape(4, 2))
"""
出力結果
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
"""
```

7.3.2 インデックス参照とスライス

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr[1]) # 出力結果: [4 5 6]
```

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr[1, 2]) # 出力結果: 6
```

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr[1,1:]) # 出力結果: [5 6]
```

7.3.3 axis

2次元配列からは**axis**という概念が重要になってきます。**axis**とは**座標軸**のようなものです。NumPyの関数の引数として**axis**を設定できる場面が多々あります。2次元配列の場合は列ごとに処理を行う軸が `axis = 0` で行ごとに行う軸が `axis = 1` ということになります。

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.sum()) # 出力結果: 21
print(arr.sum(axis = 0)) # 出力結果: [5 7 9]
print(arr.sum(axis = 1)) # 出力結果: [ 6 15]
```

7.3.4 ファンシーインデックス参照

ファンシーインデックス参照とは、インデックス参照にインデックスの配列を、用いる方法のことです。あるndarray配列から**ある特定の順序で行を抽出**するには、**その順番を示す配列**をインデックス参照として渡せば良いです。

```
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
print(arr[[3, 2, 0]]) # arr[3, 2, 0]でないことに注意
"""
出力結果
[[7 8]
 [5 6]
 [1 2]]
"""
```

7.3.5 転置行列

行列において**行と列を入れ替える事を転置**と言います。

```
arr = np.arange(10).reshape(2, 5)
print(arr.T) # print(np.transpose(arr))
"""
出力結果
[[0 5]
 [1 6]
 [2 7]
 [3 8]
 [4 9]]
"""
```

7.3.6 ソート

```
arr = np.array([[8, 4, 2], [3, 5, 1]])
print(arr.argsort()) # 出力結果 [[2 1 0] [2 0 1]]
print(np.sort(arr)) # 出力結果 [[2 4 8] [1 3 5]]
arr.sort(1) # 行でソート
print(arr) # 出力結果 [[2 4 8] [1 3 5]]
```

7.3.7 行列計算

行列計算をするための関数には、2つの行列の行列積を返す`np.dot(a, b)`関数やノルムを返す`np.linalg.norm(a)`関数等があります。

```
arr = np.array(9).reshape(3, 3)
print(np.dot(arr, arr))
"""
出力結果
[[ 15  18  21]
 [ 42  54  66]
 [ 69  90 111]]
"""
vec = arr.reshape(9)
print(np.linalg.norm(vec)) # 出力結果: 14.2828568570857
```

7.3.8 統計関数 - 1

統計関数とは**ndarray**配列全体、もしくは特定の軸を中心とした数学的な処理を行う関数、またはメソッドです。

よく使われるメソッド

メソッド	機能
`mean()`	配列の要素の平均を返す
`np.average()`	同上
`np.max()`, `np.min()`	最大値・最小値を返す
`np.std()`, `np.var()`	標準偏差・分散を返す

7.3.8 統計関数 - 2

```
arr = np.arange(15).reshape(3, 5) # [[ 0  1  2  3  4] [ 5  6  7  8  9] [10 11 12 13 14]]
print(arr.mean(axis = 0)) # 出力結果: [5. 6. 7. 8. 9.]
print(arr.sum(axis = 1)) # 出力結果: [10 35 60]
print(arr.min()) # 出力結果: 0
print(arr.argmax(axis = 0)) # 出力結果: [2 2 2 2 2]
```

7.3.9 ブロードキャスト

サイズの違うNumPy配列同士の演算時に、**ブロードキャスト**という処理が自動で行われます。ブロードキャストは、2つの**ndarray**同士の演算時にサイズの小さい配列の行と列を自動で大きい配列に合わせます。2つの配列の行が一致しない時は、行の少ないほうが多いほうの数に合わせ、足りない分を既存の行からコピーします。列が一致しない場合もまた然りです。どのような配列でもブロードキャストできるわけではありませんがすべての要素に同じ処理をする時などはブロードキャスト可能になります。

```
[[0 1 2] [3 4 5]] + 1
↓ブロードキャスト
[[0 1 2] [3 4 5]] + [[1 1 1] [1 1 1]]
↓計算結果
[[1 2 3] [4 5 6]]
```