

今回も例としてNumpyのコードを書いてますが、それらは、以下のようにNumpyをインポートしていることを前提にしています。

```
import numpy as np
```

さて。

Numpy配列内の型（dtype指定）

Numpyは配列内はすべて同じ型にそろえられます。

型を指定して生成できますし、データを与えて生成すると適切なものにしてくれるみたいですが、いちおうデフォルトはpythonのfloatと互換性のある「float_」という型になります。

dtype	互換性	コメント
float_	Pythonの「float」	Numpyのデフォルト
uint	Pythonの「int」	符号なし
int_	Pythonの「int」	符号あり
bool_	Pythonの「bool」	
int8	8ビット整数	符号あり
int16	16ビット整数	符号あり
int32	32ビット整数	符号あり
int64	64ビット整数	符号あり
uint8	8ビット整数	符号なし
uint16	16ビット整数	符号なし
uint32	32ビット整数	符号なし
uint64	64ビット整数	符号なし
float16	16ビット浮動小数点数	
float32	32ビット浮動小数点数	
float64	64ビット浮動小数点数	

これ以外にもあります。

自分はこれら以外はほぼ使わないので・・・必要なら、こちらで。

Scalars — NumPy v1.20.dev0 Manual



numpy.org

Numpy配列を構築する

メソッドを使って初期値をセットしたNumpy配列を作ります。

すべてを0で初期化する

```
np.zeros((2, 2), dtype=np.float32)
```

結果は

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

すべてを1で初期化する

```
np.ones((2, 2), dtype=np.float32)
```

結果は

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

対角要素に1それ以外に0をセットして初期化する

```
np.eye(3, dtype=np.float32)
```

結果は

```
[
```

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

指定の値ですべてを埋めた配列を作る

```
np.full((2, 2), 3.14)
```

結果は

```
[[3.14 3.14] [3.14 3.14]]
```

乱数生成を使って初期化するもの

例	説明	結果
<pre>np.random.random((2, 2))</pre>	一様分布の乱数で初期化	[[0.83874852 0.25393169] [0.48973298 0.63288969]]
<pre>np.random.normal(1, 2, (2, 2))</pre>	平均 1 標準偏差 2 の正規分布	[[2.78714269 -1.10804489] [-0.13093131 1.58371071]]
<pre>np.random.randint(0, 10, (2, 2))</pre>	指定範囲の一様分布の整数	[[0 3] [5 2]]
<pre>np.random.rand(2, 2)</pre>	0.0以上1.0未満の乱数	[[0.2219319 0.16752553] [0.53437944 0.76555056]]
<pre>np.random.randn(2, 2)</pre>	平均0分散1の乱数で初期化	[[-2.27146338 -0.00306964] [-1.04215926 1.20743357]]

`np.random` のバリエーションは沢山あります。

もっと調べたい時はこちら。

Legacy Random Generation — NumPy v1.20.dev0 Manual



numpy.org

for文なんかで少数の値を使いたいとき

```
for x in np.arange(1.0, 20.0, 0.2):  
    print(x)
```

みたいに「`np.arange`」を使う。

結果は

```
1.0  
1.2  
1.4  
1.5999999999999999  
1.7999999999999998
```

みたいになります。

pythonのリストと相互変換する

pythonのlistからNumpy配列にしたり、戻したりするやつです。

```
lstbase = [[1, 2, 3], [4, 5, 6]]  
# listからNumpy配列  
all = np.array(lstbase, dtype="float64")
```

Numpy配列からlist

```
lstafter = a1.tolist()
```

リストに戻す時は、当然、Numpy配列のdtypeと互換性のあるpythonの型になります。

Numpy配列の属性を確認する

Numpy配列の形状やサイズなどの情報を取り出します。

イメージしづらいので、こんな3行2列の配列の属性を取得する例を書きます。

```
[[6.32486041 1.82922998]
 [1.91987168 0.40812655]
 [2.4821602 1.54720082]]
```

配列の属性を確認する


上記配列の名前は「a12」としています。

書き方（配列をarとする）	意味	結果
a12.shape	配列形状のタプル	(3, 2)
a12.shape[0]	2次元の場合なら「行」	3
a12.shape[1]	2次元の場合なら「列」	2
a12.ndim	配列の次元数	2
a12.dtype	配列の型	float64
a12.size	配列の合計サイズ	6 : 3×2だから
a12.itemsize	各要素のバイト数	8 (バイト)
a12.nbytes	配列の合計バイト数	48 (バイト : size×itemsize)

のように取得できます。

他にもありますが、それはこちらで。

The N-dimensional array (ndarray) — NumPy v1.20.dev0 Manual

 numpy.org

numpy.org

ndimとaxisについて

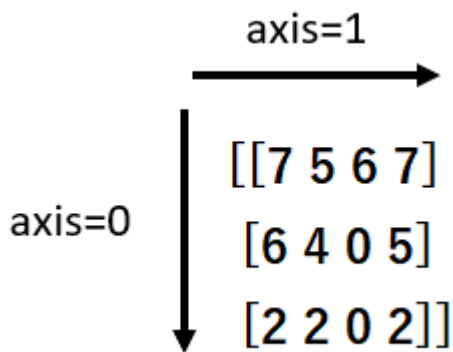
ちなみに

- ndimは「次元数」
- axisは「軸」

です。

例えば、以下のような3行4列の配列は2次元配列なので、ndimは「2」です。

axis=0を指定すると列が軸、axis=1を指定すると行が軸です。



Numpy配列のループ

Numpy配列の値をとりだして、forループを回す例です。

こんな感じの3行2列のNumpy配列を使ってやってみます。

```
[[0.83751433 0.02594981]
 [2.8441739  3.17256053]
 [0.04499033 2.14228941]]
```

上記配列は「a12」という名前で例を書いています。

データだけを取り出す場合と、インデックスとデータ両方を取り出す場合があります。

データだけを取り出すループ

```
for x in np.nditer(a12):
    print(x)
```

出力結果はこんな感じでデータだけを順番に取得します。

```
0.8375143297961567
0.025949813102027264
```

```
2.8441738981495615
3.1725605261342964
0.04499033206824665
2.142289407822542
```

インデックスとデータを取り出すパターン

インデックスは、`multi_index`で参照できます。

```
it = np.nditer(a12, flags=['multi_index'])
for x1 in it:
    print(it.multi_index, ":", x1)
```

出力結果はこんな感じになります。

```
(0, 0) : 0.8375143297961567
(0, 1) : 0.025949813102027264
(1, 0) : 2.8441738981495615
(1, 1) : 3.1725605261342964
(2, 0) : 0.04499033206824665
(2, 1) : 2.142289407822542
```

Numpy配列からのデータの取り出しとスライス

Numpy配列もPythonのリストと同じように添え字で値を参照します。

上記例で使った配列「a12」をそのまま使うなら。

```
a12[1,0]
```

とすると

```
2.8441738981495615
```

が取得できる感じで。


同様に、スライスもできて、記述方法とルールも同じです。

```
anylist[start:end:stride]
```


start : 指定した添え字を含む、省略すると「0」（つまり添え字の先頭）

end : 指定した添え字を含まない。省略すると「最後の要素まで」

stride : カウントアップする幅、省略すると「1」。-1だと逆順になる。




"BOKU"のITな日常
id:arakan_no_boku

 Hatena Blog

Pythonのシーケンススライス構文のよく使う部分をまとめておく

Pythonのスライス構文についてポイントを整理しとこうと思います。



2018-11-29 09:00

arakan-pgm-ai.hatenablog.com

Numpy配列のスライスの例です。

こういう配列データを使って例（結果列）を書いています。。

```
[[2 2 5 5]
 [4 3 0 2]
 [0 3 2 1]]
```

この配列を「a7」という名前で参照しています。

書き方	意味	結果
		[
a7[2:, :]	3行目から取り出し	[0 3 2 1]
]
a7[:, :2]	2列目までとりだし	[[2 2] [4 3] [0 3]]
a7[:, :2]	行列共に1つおきに取り出し	[[2 5] [0 2]]
a7[:, ::-1]	列を逆順にソート	[[5 5 2 2] [2 0 3 4] [1 2 3 0]]
a7[::-1, :]	行を逆順にソート	[[0 3 2 1] [4 3 0 2] [2 2 5 5]]
a7[::-1, ::-1]	行と列を共に逆順にソート	[[1 2 3 0] [2 0 3 4] [5 5 2 2]]
a7[:, 1]	2列目をとりだす	[2 3 3]
a7[1, :]	2行目をとりだす	[4 3 0 2]

Numpy配列のソート

スライスの応用の逆順ソートがでてきたので、続けて昇順にソートする方法です。

Numpyの昇順ソートは、`np.sort()`を使います。

ソート例の元にする配列は「a7」という変数名にします。

内容は以下です。

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

axisの指定無しにソートする

```
np.sort(a7)
```

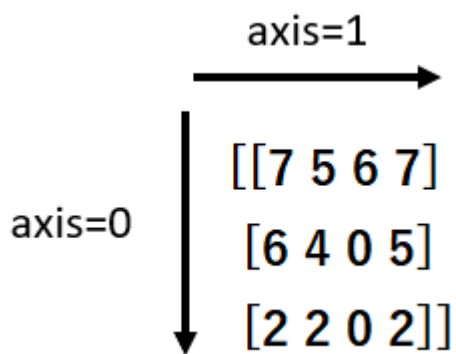
この結果はこうです。

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

行単位で小さい数値が左から並ぶようにソートされています。

axis=0を指定してソートする

axis=0は列を軸にするイメージでした。



```
np.sort(a7, axis=0)
```

この結果はこうです。

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

列単位で小さい数値順に上から並び替えられているので、行でみるとシャッフルされたような感じになってます。

axis=1を指定してソートする

axis=1は行を軸にするソートです。

```
np.sort(a7, axis=1)
```

結果はこうです。

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

行単位で左から小さい数字がならんでいます。

つまり、無指定の場合と同じjになります。

Numpy配列の形状変更

Numpy配列を処理の都合にあわせて形状変更するのはよくあります。

それらの代表的なやり方だけまとめます。

一般的な形状変換パターン

元になるNumpy配列はこんな1次元配列にします。

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

配列の名前は「aj1」としています。

これを「3×3」の二次元行列に変換するのは

```
aj33 = aj1.reshape((3, 3))
print(aj33)
```

返還後はこんな感じ

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

1次元から2次元の行へ

これも頻繁に使います。

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

を

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

にするわけです。

当然ながら、上と同じように「reshape(1, 9)」とやってもできます。

でも、スライスで「np.newaxis」を使った以下のやり方を良く見ます。

元になる1次元配列は「aj1」で参照しています。

```
ajg2 = aj1[np.newaxis, :]  
print(ajg2)
```

結果はreshape(1,3) とした場合と全く同じです。

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

1次元から列ベクトルへ

これは、ようするに。

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

を

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

```
[8]  
[9]]
```

にすることです。

当然ながら「reshape(9, 1)」とやってもできます。

でも、こちらもスライスで「np.newaxis」を使ったパターンをよく見ます。

元になる1次元配列は「aj1」で参照しています。

```
ajg2 = aj1[:, np.newaxis]  
print(ajg2)
```

結果は「(9,1)でreshape()」した場合と同じです。

多次元から1次元に

ようするに、例えば

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

を

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

に戻すことです。

flatten()を使います。

以下に一旦3×3にして、1次元に戻す例をやってます。

元の3×3配列を「aj33」として参照しています。

```
af = aj33.flatten()  
print(af)
```

flatten()の結果はこんな感じ。

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

いけてます。

NUmpy配列を転置する

転置とは、例えば「 3×4 」の配列を、「 4×3 」にするみたいに、行と列を入れ替えることです。

具体例でいえば

```
[[4 4 3 1]
 [1 0 6 1]
 [8 6 6 6]]
```

を

```
[[4 1 8]
 [4 0 6]
 [3 6 6]
 [1 1 6]]
```

にする感じです。

これは、Numpy配列を「a7」とすると

```
転置後の配列 = a7.T
```

とすればいいです。

Numpy配列の連結

例えば。

```
[[4 3 5]
 [2 5 9]
 [0 1 8]]
```

と

```
[[5 2 3]
 [4 1 6]
 [3 5 5]]
```

という2つの 3×3 配列を連結します。

便宜上、上を「g1」、下を「g2」とします。

配列を垂直（行が下に追加される感じ）で連結する

2つやり方があります。

結果はどちらも同じです。

```
# 方法1
cg1 = np.concatenate([g1, g2])
# 方法2
cg3 = np.vstack([g1, g2])
```

どちらも結果は以下のように行追加の形になります。

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

配列を水平（列が後ろに追加される感じ）で連結する。

こちらも2通りのやり方があり、結果はどちらも同じです。

```
# 方法1
cg2 = np.concatenate([g1, g2], axis=1)
# 方法2
cg4 = np.hstack([g1, g2])
```

結果は以下のように列が追加される感じになります。

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

Numpy配列の分割

連結があれば、分割がある・・・ということ。

さきほどの最終結果の

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

を分割してみます。

垂直方向（行単位でわかれていく感じ）に分割する

とりあえず3つに分けます。

2つの方法があります。

```
# 方法1
d1, d2, d3 = np.vsplit(cg2, 3)
# 方法2
ds1, ds2, ds3 = np.split(cg2, 3, axis=0)
```

どちらも結果は同じです。

注意するのは分割する数にあわせ、左辺の受取側を増やさないといけないところです。

3つを続けて表示すると。

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

水平方向（列単位でわかれていく感じ）に分割する

これも2つの方法があります。

```
# 方法1
h1, h2, h3 = np.hsplit(cg2, 3)
```

方法2

```
hs1, hs2, hs3 = np.split(cg2, 3, axis=1)
```

結果はこんな感じで列方向に3つにわかれます。

```
[[8 7]
 [7 7]
 [9 4]]

[[6 4]
 [1 5]
 [3 6]]

[[9 9]
 [7 5]
 [0 0]]
```

左辺の受取側を分割する数にあわせて増やさないといけないのは同じです。

Numpy配列同士の演算

Numpyには「ユニバーサル関数（ufunc）」と呼ばれる演算子や関数が多く実装されていて、他の方法で行うより高速に処理できるようになってます。

まずは演算子

Numpy配列同士の演算の例と計算結果を書いてます。

```
[[6 3]
 [4 6]]

[[2 2]
 [2 2]]
```

以下表の式の例は上の配列が「a」、下の配列を「b」と読み替えて結果を見てもらえるとわかりやすいです。

算術演算子	説明	式の例	結果例
+	加算	a + b	[[8 5] [6 8]]
-	減算	a - b	[[4 1] [2 4]]
*	乗算	a * b	[[12 6] [8 12]]
/	除算(結果は少数)	a / b	[[3. 1.5] [2. 3.]]

//	整数除算	a // b	[[3 1] [2 3]]
**	べき乗	a ** b	[[36 9] [16 36]]
%	余剰（あまり）	a % b	[[0 1] [0 0]]

比較演算子

演算子だけだとイメージしづらいので、以下のデータを使って例と結果を書きます。

```
[[6 3 4]
 [4 6 3]]

[[-6 3 8]
 [-4 6 4]]
```

上を「a」、下を「c」として例と結果を書いています。

算術演算子・関数	説明	式の例	結果例
==	等しい	a == c	[[False True False] [False True False]]
!=	等しくない	a != c	[[True False True] [True False True]]
<	より小さい	a < c	[[False False True] [False False True]]
<=	より小さいか等しい	a <= c	[[False True True] [False True True]]
>	より大きい	a > c	[[True False False] [True False False]]
>=	より大きいか等しい	a >= c	[[True True False] [True True False]]
np.any	条件に一致するものがひとつでもあればTrue	np.any(a == 4)	True
np.all	すべての値が条件に一致すればTrue	np.all(a == 4)	False

関数

主に集約や最大値・最小値を取得する関数です。

イメージしやすいように、以下の配列を引数にした結果を表に書いてます。

```
[[ -6  3  8]
 [ -4  6  4]]
```

上記は「d」として表現しています。

組み込み関数(例)	説明	結果
np.absolute(d)	絶対値	[[6 3 8] [4 6 4]]
np.sum(d)	配列全たの合計	11
d.sum(axis=0)	列単位の合計	[-10 9 12]
d.sum(axis=1)	行単位の合計	[5 6]
np.max(d)	配列全体の最大値	8
d.max(axis=0)	列単位の最大値	[-4 6 8]
d.max(axis=1)	行単位の最大値	[8 6]
np.argmax(d)	配列全体の最大値のインデックス	2
d.argmax(axis=0)	行単位の最大値のインデックス	[1 1 0]
d.argmax(axis=1)	列単位の最大値のインデックス	[2 1]
np.min(d)	配列全体の最小値	-6
d.min(axis=0)	列単位の最大値	[-6 3 4]
d.min(axis=1)	行単位の最大値	[-6 -4]
np.prod(d)	要素の籍を計算	13824
np.mean(d)	要素の平均値を計算	1.8333333333333333
np.median(d)	要素の中央値を計算	3.5
np.dot(a, c)	ドット積	[[-72 48 92] [-66 57 80] [-66 57 80]]

ドット積はわかりづらいので、補足のリンクです。

ベクトルの内積や行列の積を求めるnumpy.dot関数の使い方

np.dot関数は、NumPyで内積を計算する関数です。本記事では、np.dotの使い方と内積の計算について解説しています。

 deepage.net **4 users**



ユニバーサル関数は他にも沢山あります。

Universal functions (ufunc) — NumPy v1.20.dev0 Manual



numpy.org

今のところ、こんな感じです。

ファンシーインデックスや構造化配列は、今のところ自分はあまり使う機会がないのと、ページが長くなりすぎるので省いてます。

他の[まとめページ](#)と同様、ぼちぼち書き足していこうかと思います。

ではでは。