科学技術英語 英文読解課題

Ec5 24 番 平田 蓮

付録 A 発展的な NumPy

この付録では、配列計算のための NumPy ライブラリをより深く掘り下げていきます。これには ndarray 型の内部の詳細や、より高度な配列操作とそのアルゴリズムが含まれます。

この付録は雑多な内容を含んでおり続けて読む必要はありません。

A.1 ndarray オブジェクトの内部構造

NumPy の ndarray は、同じ型のデータのブロックを (連続か区切られているかにかかわらず) 多次元配列オブジェクトとして提供します。データ型 (dtype) は、データが浮動小数点、整数、真偽値、あるいはこれまで見てきた他の型のどれであるかを決定します。

ndarray を柔軟にしている理由の一つに、すべての配列オブジェクトがデータをブロック区切りでみていることがあります。例えば配列に対する参照 `arr[::2, ::-1]` がデータを一切コピーしていないことを不思議に思うかもしれません。その理由は、ndarray は単なるデータとその型の集まりではなく、配列がメモリ内をさまざまなステップサイズで移動できるようにする「ストライド」情報を持っているからです。より正確に言うと、ndarray の内部は以下のような構成になっています。

- RAM やメモリマップ内にあるデータの集まりへのポインタ
- 配列内の一定個数のデータの型
- 配列の形を表すタプル
- 各次元に沿って要素を一つ進めるために遷移するバイト数 (整数値) のタプル

図 1 は、ndarray の内部構造を示す模式図です。

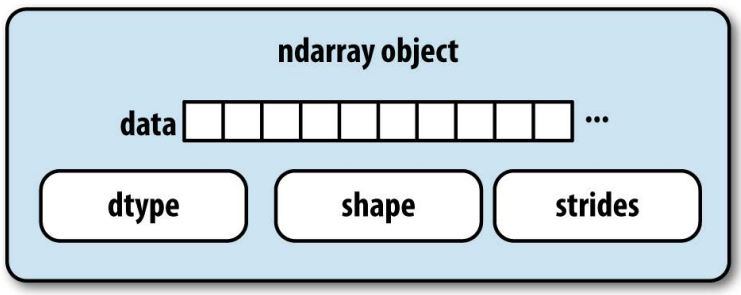

図 1　NumPy の ndarray オブジェクト

例えば、10 × 5 の配列は (10, 5) の形を持っています。

```
np.ones((10, 5)).shape # (10, 5)
```

float64(8 バイト) 型の典型的な 3×4×5 配列は、(160, 40, 8) のストライドを持っています。(一般的に特定の軸のストライドが大きいほど、その軸に沿って計算を行うコストが高くなるので、ストライドについて知っておくと便利です。)

```
np.ones((3, 4, 5), dtype=np.float64).strides # (160, 40, 8)
```

典型的な NumPy ユーザーが配列のストライドに興味を持つことは稀ですが、ストライドは 0 配列をコピー、構築するための重要な要素です。ストライドには負の値を設定することもでき、配列をメモリ上で逆向きに移動させることができます (例えば、`obj[::-1]` や `obj[:, ::-1]` のようなスライスがこれに該当します)。

NumPy の dtype の階層構造

配列の中に整数、浮動小数点数、文字列、Python オブジェクトが含まれているかどうかのチェックが必要なコードがあるかもしれません。浮動小数点数には複数の型 (float16〜float128) があるため、その配列の dtype が型一覧の中にあるかどうかをチェックするのは非常に冗長です。幸い、dtype には `np.integer` や `np.floating` などのスーパークラスがあり、`np.issubdtype()` と組み合わせて使用することができます。

```
ints = np.ones(10, dtype=np.uint16)
floats = np.ones(10, dtype=np.float32)

np.issubdtype(ints.dtype, np.integer) # True
np.issubdtype(floats.dtype, np.floating) # True
```

特定の dtype の親クラスをすべて見るには、その型の `mro` メソッドを呼び出します。

```
np.float64.mro() # [numpy.float64, numpy.floating, numpy.inexact, numpy.number, numpy.
    generic, float, object]
```

図 2 は、dtype の階層と親、サブクラスの関係を示したものです。

A.2 発展的な配列操作

配列を扱う方法には、インデックス、スライス、ブーリアンのサブセットといった洒落た方法以外のものもたくさんあります。データ分析アプリケーションの大部分は pandas の高レベル関数で処理されますが、既存のライブラリにはないデータアルゴリズムを書かなければならない場合もあるでしょう。

配列の変形

多くの場合、データを一切コピーせずに配列を他の形へ変形できます。そのためには、配列の `reshape` メソッドに新しい形を表すタプルを渡します。例えば、ある値を持つ一次元の配列を行列に並べ替えたいとします。(結果は図 3 の通りです。)

```
arr = np.arange(8) # [0, 1, 2, 3, 4, 5, 6, 7]
arr.reshape((4, 2)) # [[0, 1], [2, 3], [4, 5], [6, 7]]
```

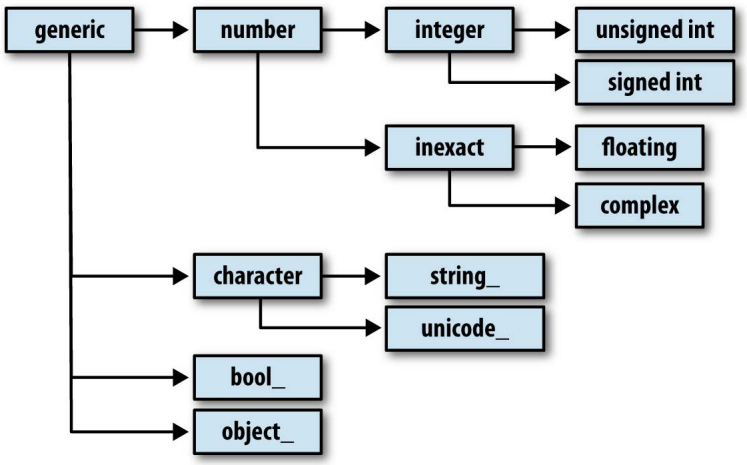

図 2　NumPy の dtype の階層構造

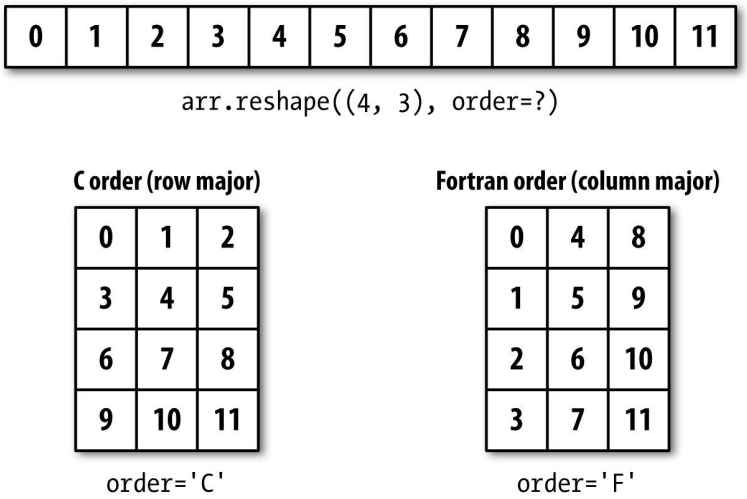

図 3　C(行優先) 順序と Fortran(列優先) 順序での変形

多次元配列は次のように変形することもできます。

```
arr.reshape((4, 2)).reshape((2, 4)) # [[0, 1, 2, 3], [4, 5, 6, 7]]
```

形を表す次元数のうち、一つは-1 にすることが可能で、その場合、その次数はデータから推測されます。

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

配列の shape 属性はタプルであるので、reshape() に渡すことも可能です。

```
other_arr = np.ones((3, 5))
other_arr.shape # (3, 5)
arr.reshape(other_arr.shape) # [[ 0, 1, 2, 3, 4], [ 5, 6, 7, 8, 9], [10, 11, 12,
    13, 14]]
```

一次元から高次元への変形の逆の操作は一般的にフラットニングやラベリングと呼ばれます。

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

ravel() は変形後の値が隣接している場合、元の値をコピーしません。flatten メソッドは常にデータのコピーを返すという点を除いて、ravel と同じ動作をします。

```
arr.flatten() # [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

データは違う順序で変形したり、ラベリングしたりすることができます。これは、NumPy の新規ユーザーにとっては少しニュアンスの異なるトピックであるため、次のサブトピックになっています。

C 順序と Fortran 順序

NumPy は、メモリ上のデータのレイアウトをコントロールし、柔軟に変更することができます。デフォルトでは、NumPy の配列は行優先に作成されます。これは空間的には、データの 2 次元配列を持っている場合、配列の各行のアイテムは隣接するメモリ位置に格納されることを意味します。歴史的な理由から、行と列の主要な順序は、それぞれ C と Fortran の順序としても知られています。FORTRAN 77 言語では、行列はすべて列優先の順序で表されます。これは通常、「C」または「F」のように設定できます (他にもよく使われるオプション「A」と「K」があります)。

配列の結合・分割

np.concatenate は、配列を受け取り、入力の軸に従って結合します。

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([[7, 8, 9], [10, 11, 12]])

np.concatenate([arr1, arr2], axis=0) # [[ 1,  2,  3],[ 4,  5,  6],[ 7,  8,  9],[10, 11, 12]]
np.concatenate([arr1, arr2], axis=1) # [[ 1,  2,  3,  7,  8,  9],[ 4,  5,  6, 10, 11, 12]]
```

vstack や hstack などの便利な関数があります。これらは次のような働きをします。

```
np.vstack((arr1, arr2)) # [[ 1,  2,  3],[ 4,  5,  6],[ 7,  8,  9],[10, 11, 12]]
np.hstack((arr1, arr2)) # [[ 1,  2,  3,  7,  8,  9],[ 4,  5,  6, 10, 11, 12]]
```

split は逆に、配列を入力の軸に従って分割します。

```
arr = np.random.randn(5, 2) # [[-0.2047,  0.4789],[-0.5194, -0.5557],[ 1.9658,  1.3934],[ 0.0929,  0.2817],[ 0.769 ,  1.2464]]
np.split(arr, [1, 3]) # [[-0.2047,  0.4789]], [[-0.5194, -0.5557],[ 1.9658,  1.3934]], [[ 0.0929,  0.2817],[ 0.769 ,  1.2464]]
```

引数の [1, 3] は、配列を分割する添字を示しています。

関連するすべての連結・分割関数の一覧は表 1 を参照してください。中には非常に汎用性の高い concatenate の利便性のためだけに提供されているものもあります。

表 1　配列を連結・分割する関数

関数	説明
concatenate	一般的な関数。一方向に複数の配列を結合する。
vstack, row_stack	行方向に配列を結合する。
hstack	列方向に配列を結合する。
column_stack	hstack と同じだが、最初に一次元配列を二次元列ベクトルに変換する。
dstack	奥行き方向に配列を結合する。
split	特定の軸方向に、指定した添字で配列を分割する。
hsplit, vsplit	それぞれ split を行方向、列方向に行う。

おしゃれな配列参照

整数の配列を用いて、少し変わった参照を行うことができます。

```
arr = np.arange(10) * 100 # [0, 100, 200, 300, 400, 500, 600, 700, 800, 900]
inds = [7, 1, 2, 6]

arr[inds] # [700, 100, 200, 600]
```

似た働きをする関数、take と put があります。

```
arr.take(inds) # [700, 100, 200, 600]
arr.put(inds, 42) # [0, 42, 42, 300, 400, 500, 42, 42, 800, 900]
```

put は軸の引数を受け取らず, C 順に一次元化された配列にインデックスを付けます。このように、他の添字配列を使用して要素を設定する必要がある場合は多くの場合、この参照方法を使用するのが最も簡単です。

A.3 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での演算方法を表現したものです。これは強力な機能ですが、経験豊富なユーザでも混乱することがあります。最も簡単なブロードキャストの例は、スカラ値と配列の組み合わせです。

```
r = np.arange(5) # array([0, 1, 2, 3, 4])
arr * 4 # array([ 0,  4,  8, 12, 16])
```

これは、スカラー値である 4 が乗算の際に他のすべての要素にブロードキャストされたことを示しています。例えば、配列の各列の平均値を減算することで、列を小さくすることができます。これは、非常に簡単です。

```
r = np.random.randn(4, 3)
arr.mean(0) # array([-0.3928, -0.3824, -0.8768])
```

```
3 demeaned = arr - arr.mean(0) # array([[ 0.3937,  1.7263,  0.1633],[-0.4384,
      -1.9878, -0.9839],[-0.468 ,  0.9426, -0.3891],[ 0.5126, -0.6811,  1.2097]])
4 demeaned.mean(0) # array([-0.,  0., -0.])
```

A.4 発展的な ufunc の使い方

多くの NumPy ユーザーは、ユニバーサル関数が提供する高速な要素間演算のみを利用するでしょうが、ループなしでより簡潔なコードを書くのに役立つ追加機能がいくつかあります。

ufunc インスタンスメソッド

NumPy のバイナリ関数には、ある種の特殊なベクトル演算を行うための特別なメソッドがあります。reduce は、1 つの配列を受け取り、一連のバイナリ操作を実行することで、その値を集約します。例えば、配列の要素の合計を求めるには、np.add.reduce を使用します。

```
1 arr = np.arange(10)
2 np.add.reduce(arr) # 45
3 arr.sum() # 45
```

開始値（add の場合は 0）は ufunc に依存します。軸が渡された場合、その軸に沿って演算が実行されます。これにより、ある種の値を簡潔に答えることができます。具体的な例としては、np.logical_and を使って配列の各行の値がソートされているかどうかを調べることができます。(ここで、logical_and.reduce と all は同じ働きをします。)

```
1 np.random.seed(12346)
2 arr = np.random.randn(5, 5)
3 arr[::2].sort(1)
4 arr[:, :-1] < arr[:, 1:] # array([[ True,  True,  True,  True],[False,  True, False,
      False],[ True,  True,  True,  True],[ True, False,  True,  True],[ True,  True,
      True,  True]], dtype=bool)
5
6 np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=1) # array([ True, False,  True,
       False,  True], dtype=bool)
```

A.5 構造化配列と記憶配列

これまで、ndarray は同種のデータコンテナであるように触れてきました。つまり、各要素が dtype によって決められた同じバイト数を占めるメモリブロックを表しています。これでは一見、異質なデータや表形式のデータを表現できないように見えます。構造化配列とは、各要素が C 言語の構造体（「構造化」という名前の由来）や、複数の名前付きフィールドを持つ SQL テーブルの行に相当すると考えられる配列です。

構造化されたデータタイプを指定するにはいくつかの方法があります (オンラインの NumPy ドキュメントを参照してください)。典型的な方法の 1 つは、(field_name, field_data_type) のタプルのリストとして指定する方法です。この場合、配列の要素はタプルのようなオブジェクトで、その要素は辞書型のようにアクセスできます。

フィールド名は、dtype.names 属性に格納されています。構造化配列のフィールドにアクセスすると、データのストライドビューが返されるため、何もコピーされません。

なぜ構造化配列を使うのか

NumPy の構造化配列は、pandas の DataFrame などと比較して、比較的低レベルなツールです。構造化配列は、メモリブロックを任意の複雑な入れ子構造の列を持つ表形式の構造として解釈する手段を提供します。構造化配列の各要素は、メモリ上では固定のバイト数で表現されるため、構造化配列のもう一つの一般的な用途として、固定長レコードのバイトストリームとしてデータファイルを記述することがありますが、これは C や C++ コードでデータを並べる一般的な方法であり、産業界のレガシーシステムでしか見られません。ファイルのフォーマット（各レコードのサイズ、各要素の順序、バイトサイズ、データタイプ）がわかっていれば、`np.fromfile` でデータをメモリに読み込むことができます。このような特殊な使い方は本書では説明できませんが、このようなことが可能であることを知っておくとよいでしょう。