NumPy解説：Pythonの数値計算を支える強力なライブラリの実践ガイド

はじめに

本レポートは、Pythonにおける数値計算の基盤となるライブラリ、NumPy（ナムパイ）の基本的な使い方を解説することを目的としています。特に、NumPyの中核である多次元配列オブジェクトndarrayの作成、属性確認、基本的な演算、要素へのアクセス、形状変更、そしてブロードキャスティングといった、NumPyを使いこなす上で不可欠な機能に焦点を当てます。
対象読者としては、Pythonでの数値計算やデータ分析をこれから始める学生やプログラマー、あるいはNumPyの基礎を再確認したい方を想定しています。各機能について、具体的なコード例を交えながら、明確かつ実践的に解説を進めていきます。

1. NumPy入門：科学技術計算Pythonの中核

NumPyとは何か、なぜ重要なのか

NumPy（Numerical Python）は、Pythonで科学技術計算を行うための基本的なパッケージです。その最も重要な貢献は、効率的な多次元配列オブジェクトndarray（N-dimensional array）の提供にあります。
Python標準のリストは非常に柔軟ですが、数値計算、特に大量のデータを扱う場合にはいくつかの点で非効率です。リストの要素はメモリ上に散在するオブジェクトへのポインタであり、演算時には要素ごとの型チェックが必要になります。一方、NumPyのndarrayは、すべての要素が同じデータ型であり、メモリ上で連続的に配置されるように設計されています``。
このデータ構造の特性により、NumPyの演算は、高度に最適化され、プリコンパイルされたC言語のコード（ベクトル化、ufuncと呼ばれる仕組み）によって実行されます``。これにより、Pythonのループ処理に伴うオーバーヘッドや型チェックが回避され、特に大規模なデータセットに対する数値演算において、Pythonリストと比較して劇的な速度向上が実現されます。この計算効率こそが、NumPyがデータサイエンス、機械学習、統計分析など、現代の多くの計算科学分野で不可欠な基盤ライブラリとなっている最大の理由です。
さらに、NumPyはSciPy（高度な科学計算）、Pandas（データ分析）、Matplotlib（グラフ描画）、Scikit-learn（機械学習）といった、他の多くの重要な科学技術計算ライブラリの基盤としても機能しています。これらのライブラリは、内部でNumPyのndarrayを利用してデータを効率的に処理しています。

N次元配列（ndarray）の力

ndarrayは、同じ型の値からなるグリッド（格子状のデータ構造）であり、非負整数のタプルによってインデックス付けされます。この「次元」は「軸（axis）」とも呼ばれます。1次元配列はベクトル、2次元配列は行列に相当し、NumPyはさらに高次元の配列も扱うことができます。
ndarrayの重要な特性の一つは、前述の通り、要素がすべて**固定されたデータ型（dtype）**を持つことです。これはパフォーマンスとメモリ効率の鍵となります。Pythonリストが各要素の型情報を個

別に保持する必要があるのに対し、ndarrayは配列全体の型情報を一つ持つだけで済み、要素自体は生の数値データとして格納されるため、メモリ使用量が削減されます ``。特に、単純な数値型の要素を大量に格納する場合、この差は顕著になります。
また、データがメモリ上で連続的に配置されることは、CPUキャッシュの効率的な利用（キャッシュローカリティの向上）にも繋がり、計算速度の向上に寄与します。
NumPyの多くの操作は**要素ごと（element-wise）**に行われます。これにより、ループを明示的に書くことなく、配列全体に対する演算を簡潔かつ効率的に記述できます。この「ベクトル化」された記述スタイルは、NumPyプログラミングの基本となります（詳細は後述）。

2. はじめてのNumPy：配列の作成

NumPyで計算を行うためには、まずndarrayオブジェクトを作成する必要があります。慣例として、NumPyはnpという別名でインポートされます。

Python

```
import numpy as np
```

既存のデータ（Pythonリスト、タプル）からの作成 - np.array()

最も基本的な配列作成方法は、Pythonのリストやタプルをnp.array()関数に渡すことです ``。

Python

```
# 1次元配列の作成
list_a =
arr_a = np.array(list_a)
print(arr_a)
# 出力: [1 2 3 4 5]
```

ネストされたリスト（リストのリスト）を渡すことで、多次元配列を作成できます ``。

Python

```
# 2次元配列（行列）の作成
list_b = [[1.0, 2.0], [3.0, 4.0]]
arr_b = np.array(list_b)
print(arr_b)
# 出力:
# [[1. 2.]
```

[3. 4.]]

np.array()は、入力されたデータから自動的にデータ型（dtype）を推測します。例えば、整数のみのリストからは整数型（int64など）、浮動小数点数が含まれていれば浮動小数点数型（float64など）の配列が作成されます。dtype引数を使って、データ型を明示的に指定することも可能です ``。

Python

```
# データ型を float64 に指定して作成
list_c =
arr_c = np.array(list_c, dtype=np.float64)
print(arr_c)
print(arr_c.dtype)
# 出力:
# [1. 2. 3.]
# float64
```

ここで注意すべき点は、np.array()はデフォルトで入力データのコピーを作成するということです。つまり、元のPythonリストを変更しても、作成されたNumPy配列には影響しません ``。

Python

```
original_list =
array_from_list = np.array(original_list)

# 元のリストを変更
original_list = 99
print(f"元のリスト: {original_list}")
print(f"作成された配列: {array_from_list}")
# 出力:
# 元のリスト:
# 作成された配列: [10 20 30]
```

このコピー動作は、元のデータソースからの独立性を保証するため、一般的には安全な挙動です。しかし、非常に大きなリストから配列を作成する場合、メモリ使用量が一時的に倍増する可能性がある点には留意が必要です。

配列の初期化（プレースホルダー）

特定の値を入力するのではなく、特定の形状を持つ配列を初期値（例えば、すべて0や1）で埋めて作成したい場合があります。これは、計算結果を格納するためのスペースを事前に確保する際などに便利です。

- np.zeros(shape, dtype=float): 指定された形状（shape、タプルで指定）を持ち、すべての要素が0で埋められた配列を作成します。dtypeはオプションで、デフォルトはfloat64です ``。
  Python
  zeros_arr = np.zeros((2, 3)) # 2行3列のゼロ行列
  print(zeros_arr)
  # 出力:
  # [[0. 0. 0.]
  # [0. 0. 0.]]

- np.ones(shape, dtype=float): すべての要素が1で埋められた配列を作成します ``。
  Python
  ones_arr = np.ones((3, 2), dtype=np.int32) # 3行2列、int32型の1行列
  print(ones_arr)
  # 出力:
  # [[1 1]
  # [1 1]
  # [1 1]]

- np.full(shape, fill_value, dtype=None): 指定されたfill_valueで全ての要素を埋めた配列を作成します ``。
  Python
  full_arr = np.full((2, 2), 99) # 2x2の、要素が全て99の配列
  print(full_arr)
  # 出力:
  # [[99 99]
  # [99 99]]

- np.empty(shape, dtype=float): 指定された形状の配列を作成しますが、要素の初期化を行いません ``。配列の要素には、そのメモリ領域に以前存在していた任意の値（ゴミデータ）が含まれる可能性があります。zerosよりも高速ですが、使用前に必ず全ての要素に値を代入する必要があります。
  Python
  empty_arr = np.empty((2, 2))
  print(empty_arr) # 出力は実行ごとに異なる可能性がある
  # 例:
  # [[6.95094549e-310 6.95094549e-310]
  # [1.31659352e-316 4.48896218e-317]]

np.emptyがnp.zerosより高速なのは、メモリを確保するだけで、各要素に0を書き込むステップを省略するためです。配列作成後すぐに全要素を計算結果などで上書きする予定がある場合、emptyを使うことで、特に巨大な配列を頻繁に作成する際のわずかながらパフォーマンス向上に繋がることがあります。ただし、初期化されていない値を読み込むと予期せぬ動作を引き起こすため、注意が必要です。

連番配列の生成

等差数列のような、規則的な値を持つ配列を生成する関数も用意されています。

- np.arange([start,] stop[, step,], dtype=None): Pythonのrange関数に似ていますが、ndarrayを返します。指定された間隔で等間隔の値を生成します ``。stopで指定された値は含まれません。
  Python
  arange_arr1 = np.arange(10) # 0から9までの整数配列
  print(arange_arr1)
  # 出力: [0 1 2 3 4 5 6 7 8 9]

  arange_arr2 = np.arange(2, 10, 2) # 2から始まり10未満、ステップ2
  print(arange_arr2)
  # 出力: [2 4 6 8]

- np.linspace(start, stop, num=50, endpoint=True, dtype=None): 指定された区間 (startからstopまで) を、指定された要素数 (num) で等分割する値を生成します ``。重要な点として、デフォルト (endpoint=True) ではstopの値が含まれます。
  Python
  linspace_arr = np.linspace(0, 1, 5) # 0から1までを5つの要素で等分割
  print(linspace_arr)
  # 出力: [0.   0.25 0.5 0.75 1.]

arangeとlinspaceの使い分けは重要です。arangeはステップ幅を指定するのに対し、linspaceは要素数を指定します。特に浮動小数点数の区間を扱う場合、arangeは浮動小数点数の精度限界により、期待した要素数にならない、あるいはstop値が含まれるかどうかが不確実になることがあります ``。一方、linspaceはstart, stop, numからステップ幅を計算するため、指定した要素数を正確に生成し、endpoint=Trueであればstop値を確実に含みます。関数のサンプリング点やプロット用の座標生成など、区間内の正確な要素数が重要な場合には、linspaceの方がよりロバストで予測可能な結果を与えるため、一般的に好まれます。

(オプション)単位行列と乱数配列

- np.eye(N, M=None, k=0, dtype=float): 2次元の単位行列(対角成分が1で、それ以外が0の正方行列)を作成します ``。
  Python
  identity_matrix = np.eye(3)
  print(identity_matrix)
  # 出力:
  # [[1. 0. 0.]
  # [0. 1. 0.]
  # [0. 0. 1.]]

- np.random モジュール: NumPyには乱数生成のための強力なサブモジュール random があ

ります。例えば、np.random.rand(d0, d1,..., dn)（0から1までの一様乱数）、np.random.randn(d0, d1,..., dn)（標準正規分布に従う乱数）、np.random.randint(low, high=None, size=None, dtype=int)（指定範囲の整数乱数）などがあります ``。乱数の詳細はこのレポートの範囲外ですが、その存在を知っておくと便利です。

```
Python
random_arr = np.random.rand(2, 2) # 2x2の0-1一様乱数配列
print(random_arr)
# 例:
# [[0.123 0.456]
#  [0.789 0.012]]
```

まとめ：主な**NumPy**配列作成関数

関数	目的	例（構文）
np.array(list)	Pythonリスト等から配列を作成（コピー）	np.array()
np.zeros(shape)	指定形状のゼロ配列を作成	np.zeros((2, 3))
np.ones(shape)	指定形状の要素が1の配列を作成	np.ones((3, 1))
np.full(shape, val)	指定形状の、指定値で埋められた配列を作成	np.full((2, 2), 7)
np.empty(shape)	指定形状の未初期化配列を作成（高速）	np.empty((2, 2))
np.arange(stop)	連番配列を作成（stopを含まない）	np.arange(5)
np.linspace(s, e, n)	等間隔配列を作成（n個、eを含む）	np.linspace(0, 10, 5)
np.eye(N)	単位行列を作成	np.eye(3)
np.random.rand(d...)	指定形状の乱数配列を作成（一様分布）	np.random.rand(2, 2)

この表は、本セクションで紹介した主要な配列作成関数をまとめたものです。それぞれの関数の目的と基本的な使い方を比較することで、状況に応じた適切な関数を選択する助けとなります。

3. 配列の理解：主要な属性

配列を作成したら、その性質（次元数、形状、要素数、データ型など）を理解することが重要です。これらの情報は、配列オブジェクトの属性（メソッドではないため、括弧()は付けずにアクセスします）を通じて取得できます。

- 次元数 **(ndarray.ndim)**: 配列の軸（次元）の数を返します ``。
  ```
  Python
  arr_3d = np.zeros((2, 3, 4)) # 形状が (2, 3, 4) の3次元配列
  ```

print(arr_3d.ndim)
# 出力: 3

- 形状 **(ndarray.shape)**: 配列の各次元のサイズを示す整数のタプルを返します。このタプルの長さが、配列の次元数 (ndim) になります ``。
  Python
  arr_2d = np.array([, ,]) # 3行2列の配列
  print(arr_2d.shape)
  # 出力: (3, 2)

  shape属性は、おそらく最も重要な属性です。配列の要素がどのように構成されているかを示し、他の配列との演算（要素ごとの加算、行列積など）、ブロードキャスティング、連結などの互換性を決定します。形状の不一致はNumPyにおける一般的なエラーの原因であるため、開発やデバッグ中に.shapeを確認する習慣は、特に多次元配列を扱う上で非常に重要です。
- 総要素数 **(ndarray.size)**: 配列に含まれる全要素の数を返します。これは、shapeタプルの全要素の積と等しくなります ``。
  Python
  arr_1d = np.arange(10)
  print(arr_1d.size)
  # 出力: 10

  arr_zeros = np.zeros((3, 4))
  print(arr_zeros.size)
  # 出力: 12

- データ型 **(ndarray.dtype)**: 配列の要素のデータ型を示すオブジェクトを返します ``。
  Python
  arr_int = np.array()
  print(arr_int.dtype)
  # 出力: int64 (環境により int32 の場合もある)

  arr_float = np.array([1.0, 2.0])
  print(arr_float.dtype)
  # 出力: float64

  よく使われるデータ型には、int32, int64（整数）、float32, float64（浮動小数点数）、bool（ブール値）、complex（複素数）、object（Pythonオブジェクト）などがあります。
  dtypeは、配列が使用するメモリ量、計算の精度、そして場合によっては計算速度に直接影響します ``。例えば、float64（8バイト/要素）の代わりにfloat32（4バイト/要素）を使用すると、メモリ使用量を半分に削減できますが、計算精度は低下します。整数型を使用すると、小数点以下の計算はできません。大規模なデータセットを扱う場合、メモリフットプリントを削減するためにfloat32を選択することが有効な場合がありますが、計算内容によってはfloat64の精度が必要になることもあります。また、特定のハードウェア（GPUなど）では、float32での計算が

高速な場合もあります。したがって、dtypeの選択は、アプリケーションの要件に応じてメモリ、精度、速度のトレードオフを考慮して行う必要があります。

- （オプション）その他の属性:
    - ndarray.itemsize: 配列の各要素が占めるバイト数を返します。
    - ndarray.nbytes: 配列全体が占める総バイト数を返します (size * itemsize と等価)。``` Python
arr = np.array([1.0, 2.0, 3.0], dtype=np.float64)
print(f"Item size: {arr.itemsize} bytes") # float64 は 8 バイト
print(f"Total bytes: {arr.nbytes} bytes") # 3要素 * 8バイト = 24 バイト
出力:
Item size: 8 bytes
Total bytes: 24 bytes

arr_f32 = arr.astype(np.float32) # float32 に型変換
print(f"Item size (float32): {arr_f32.itemsize} bytes") # float32 は 4 バイト
print(f"Total bytes (float32): {arr_f32.nbytes} bytes") # 3要素 * 4バイト = 12 バイト
出力:
Item size (float32): 4 bytes
Total bytes (float32): 12 bytes
```

### まとめ：主要な**NumPy**配列属性

| 属性 | 説明 | 出力例 **(例: arr = np.zeros((2,3)))** |
|---|---|---|
| arr.ndim | 配列の次元数 | 2 |
| arr.shape | 各次元のサイズ（タプル） | (2, 3) |
| arr.size | 全要素数 | 6 |
| arr.dtype | 要素のデータ型 | float64 |
| arr.itemsize | 1要素あたりのバイト数 | 8 (float64の場合) |
| arr.nbytes | 配列全体の総バイト数 | 48 (6要素 * 8バイト) |

この表は、配列の基本的な特性を把握するために頻繁に使用される属性をまとめたものです。これらの属性を理解し、適切に確認することは、NumPyを効果的に使用するための第一歩です。

## 4. 計算の実行：配列の算術演算と数学関数

NumPyの主な強みの一つは、配列に対する高速なベクトル化演算です。

### 要素ごと（**Element-wise**）の演算

標準的な算術演算子（+, -, *, /, **（べき乗）など）は、同じ形状を持つ配列間で要素ごとに作用します``` 。

Python
```

```
a = np.array()
b = np.array()

print(f"a + b = {a + b}")    # 要素ごとの加算
# 出力: a + b = [5 7 9]

print(f"a * b = {a * b}")    # 要素ごとの乗算
# 出力: a * b = [ 4 10 18]

print(f"a ** 2 = {a ** 2}")  # 要素ごとのべき乗
# 出力: a ** 2 = [1 4 9]
```

この要素ごとの演算は、多次元配列に対しても同様に適用されます ``。

```
Python


matrix_a = np.array([, ])
matrix_b = np.array([, ])

print(f"Matrix A + Matrix B:\n{matrix_a + matrix_b}")
# 出力:
# Matrix A + Matrix B:
# [[ 6  8]
#  [10 12]]
```

このNumPyの振る舞いは、Pythonのリストで同様の操作を行う場合と対照的です。リストで要素ごとの演算を行うには、明示的なforループが必要です（例: c = [a[i] + b[i] for i in range(len(a))]）。NumPyでは、c = a + b と書くだけで同じ結果が得られます。NumPyの内部では、この演算が最適化された単一のC言語ループで実行されるため、各要素に対するPythonインタープリタのオーバーヘッドがありません ``。この「ベクトル化」（配列全体に対する操作を一度に記述し、コンパイル済みコードで実行する）アプローチは、ndarrayの特性（均一なデータ型、連続したメモリ配置）によって可能になり、数値計算の効率を飛躍的に向上させる核心的な原理です。

ユニバーサル関数 (ufunc)

NumPyは、ndarrayに対して要素ごとに作用するユニバーサル関数（**ufunc**）を多数提供しています。これらには、基本的な数学関数が含まれます。

- np.sqrt(): 要素ごとの平方根 ``。
  Python
  ```
  arr = np.array()
  print(np.sqrt(arr))
  # 出力: [1. 2. 3.]
  ```

- np.exp(): 要素ごとの指数関数 (e^x) ``。
  Python
  print(np.exp(arr))
  # 出力: [2.71828183e+00 5.45981500e+01 8.10308393e+03]

- np.sin(), np.cos(), np.log(): 要素ごとの三角関数、対数関数など ``。
  Python
  angles = np.array([0, np.pi/2, np.pi])
  print(np.sin(angles))
  # 出力: [0.0000000e+00 1.0000000e+00 1.2246468e-16] (ほぼ[0. 1. 0.])

実は、前述の算術演算子（+, *など）も、内部的にはufunc（それぞれnp.add, np.multiplyなど）として実装されています。
ufuncは、算術演算であれ数学関数であれ、要素ごとの操作に対して一貫したAPIを提供します。これにより、NumPyの学習と使用が容易になります。一つのパターン（関数を配列に適用すると要素ごとに作用する）が多くの操作に適用されるため、予測可能性が高まります。さらに、ufuncの仕組みは拡張可能であり、SciPyのようなNumPy上に構築されたライブラリは、特殊関数など、独自のufuncを追加してNumPy配列とシームレスに連携させることができます。

（オプション）集約関数

配列全体または特定の軸に沿って値を集約する関数もよく使われます。

- np.sum(): 全要素の合計 ``。
- np.min(), np.max(): 最小値、最大値 ``。
- np.mean(), np.std(), np.var(): 平均値、標準偏差、分散 ``。

これらの関数は、配列オブジェクトのメソッドとしても利用できます（例: arr.sum(), arr.min()）。

Python

```
data = np.arange(5) # [0 1 2 3 4]
print(f"Sum: {np.sum(data)}")      # or data.sum()
print(f"Min: {data.min()}")
print(f"Mean: {data.mean()}")
# 出力:
# Sum: 10
# Min: 0
# Mean: 2.0
```

多次元配列の場合、これらの集約関数はaxis引数を取ることができます。axisは、操作を実行する軸を指定します。例えば、2次元配列（行列）の場合：

- axis=0: 各列に沿って（縦方向に）操作を実行します。結果の次元数は1つ減ります。
- axis=1: 各行に沿って（横方向に）操作を実行します。

axisパラメータの理解は、行列演算やデータ分析（例：各特徴量（列）の平均値を計算する）においてNumPyを最大限に活用するために不可欠です ``。

Python

```
matrix = np.array([, ])
print(f"元の行列:\n{matrix}")

# 各列の合計 (axis=0)
col_sum = matrix.sum(axis=0)
print(f"列ごとの合計 (axis=0): {col_sum}") # shape: (3,)
# 出力: 列ごとの合計 (axis=0): [5 7 9]

# 各行の合計 (axis=1)
row_sum = matrix.sum(axis=1)
print(f"行ごとの合計 (axis=1): {row_sum}") # shape: (2,)
# 出力: 行ごとの合計 (axis=1): [ 6 15]
```

5. データへのアクセス：インデックス参照とスライシング

配列の特定の部分（要素や部分配列）にアクセスすることは、基本的な操作です。

単一要素へのアクセス（インデックス参照）

- **1**次元配列: Pythonのリストと同様に、ゼロベースの整数インデックスを角括弧内に指定します。負のインデックスは末尾からの位置を示します (`-1`が最後の要素)。
  Python
  ```
  arr_1d = np.arange(10) # [0 1 2 3 4 5 6 7 8 9]
  print(arr_1d)  # 3番目の要素
  # 出力: 2
  print(arr_1d[-1])  # 最後の要素
  # 出力: 9
  ```

- 多次元配列: 各次元のインデックスをカンマ区切りのタプルとして、一つの角括弧内に指定します (`[行, 列,...]`)。
  Python
  ```
  arr_2d = np.array([, ])
  print(arr_2d)  # 1行目(index 0), 2列目(index 1)の要素
  # 出力: 2
  print(arr_2d[1, -1]) # 2行目(index 1), 最後の列(index -1)の要素
  # 出力: 6
  ```

多次元配列のインデックス参照では、arr_2dのように角括弧を連鎖させることも可能ですが、arr_2dというタプル形式の方がNumPyでは一般的であり、より効率的です。arr_2dは、まず1行目を取得する一時的な配列を作成し、次にその一時配列から要素を取得するという2段階の操作になります。一方、arr_2dは単一の最適化されたインデックス参照操作で要素に直接アクセスします。特に値の代入 (arr_2d = 100) を行う際には、タプル形式が推奨されます。

基本的なスライシング（部分配列の抽出）

スライシングは、配列の一部を切り出して部分配列（サブアレイ）を取得する機能です。コロン:を使った start:stop:step 形式で範囲を指定します。

- start: 開始インデックス（含まれる）。省略すると先頭 (0) から。
- stop: 終了インデックス（含まれない）。省略すると末尾まで。
- step: ステップ幅（増分）。省略すると1。
- **1**次元配列:
  Python
  ```
  arr_1d = np.arange(10) # [0 1 2 3 4 5 6 7 8 9]
  print(arr_1d[2:5])   # index 2 から 5 の手前まで (index 2, 3, 4)
  # 出力: [2 3 4]
  print(arr_1d[:4])    # 先頭から index 4 の手前まで (index 0, 1, 2, 3)
  # 出力: [0 1 2 3]
  print(arr_1d[5:])    # index 5 から末尾まで
  # 出力: [5 6 7 8 9]
  print(arr_1d[:])     # 全要素 (配列のコピーではない、後述)
  # 出力: [0 1 2 3 4 5 6 7 8 9]
  ```

- 多次元配列: 各次元に対して、インデックス参照またはスライシングをカンマ区切りで適用します。
  Python
  ```
  arr_2d = np.arange(12).reshape((3, 4)) # 3行4列の配列
  # [[ 0  1  2  3]
  #  [ 4  5  6  7]
  #  [ 8  9 10 11]]

  # 最初の2行 (index 0, 1) と、列 index 1
  print(arr_2d[0:2, 1])
  # 出力: [1 5]

  # 行 index 1 と、列 index 1 から 3 の手前まで (index 1, 2)
  print(arr_2d[1, 1:3])
  # 出力: [5 6]

  # 最初の2行 (index 0, 1) と、列 index 1 から 3 の手前まで (index 1, 2)
  print(arr_2d[:2, 1:3])
  # 出力:
  ```

```
# [[1 2]
#  [5 6]]

# 全ての行と、列 index 1 (つまり、2列目全体)
print(arr_2d[:, 1])
# 出力: [1 5 9]
```

: を単独で使うと、その次元全体を選択することになります ``。

ステップ付きスライシング

stepを指定することで、要素を一定間隔で飛び飛びに取得できます ``。

Python

```
arr_1d = np.arange(10) # [0 1 2 3 4 5 6 7 8 9]

# 全要素を、ステップ 2 で取得 (偶数インデックス)
print(arr_1d[::2])
# 出力: [0 2 4 6 8]

# index 1 から 7 の手前まで、ステップ 2 で取得
print(arr_1d[1:7:2])
# 出力: [1 3 5]
```

stepに負の値を指定すると、逆順に要素を取得できます。特に [::-1] は配列全体を逆順にする簡単な方法です ``。

Python

```
print(arr_1d[::-1])
# 出力: [9 8 7 6 5 4 3 2 1 0]
```

スライスはコピーではなくビュー（**View**）である

これはNumPyにおける非常に重要な概念です。配列のスライス操作は、多くの場合、元の配列データの**ビュー（view）**を返します。ビューは元の配列と同じデータを参照しているため、ビューを変更すると元の配列も変更されます ,。

Python

```
original_arr = np.arange(5) # [0 1 2 3 4]
print(f"Original array (before): {original_arr}")

# スライス (ビュー) を取得
slice_view = original_arr[1:4] # [1 2 3]
print(f"Slice (view): {slice_view}")

# スライス (ビュー) の要素を変更
slice_view = 99
print(f"Slice (after modification): {slice_view}")
print(f"Original array (after modification): {original_arr}") # 元の配列も変更されている！
# 出力:
# Original array (before): [0 1 2 3 4]
# Slice (view): [1 2 3]
# Slice (after modification): [99  2  3]
# Original array (after modification): [ 0 99  2  3  4]
```

この挙動は、コピーを作成するPythonリストのスライシングとは異なります。NumPyのビューの仕組みは、大規模な配列を扱う際にデータの複製を避けるため、メモリ効率が非常に高いです。しかし、意図しない副作用を防ぐためには注意が必要です。もし、元の配列に影響を与えない独立したコピーが必要な場合は、明示的に .copy() メソッドを使用する必要があります ``。

Python

```
original_arr = np.arange(5) # [0 1 2 3 4]
slice_copy = original_arr[1:4].copy() #.copy() を使ってコピーを作成

slice_copy = 999

print(f"Slice (copy): {slice_copy}")
print(f"Original array (should be unchanged): {original_arr}") # 元の配列は変更されない
# 出力:
# Slice (copy): [999   2   3]
# Original array (should be unchanged): [0 1 2 3 4]
```

スライスがビューであるかコピーであるかは常に意識し、必要に応じて .copy() を使うことが、バグを防ぐ上で重要です。

6. 配列に対する条件付きロジック：ブールインデックス参照

配列の要素を、その位置ではなく、特定の条件に基づいて選択したい場合があります。これを実現するのがブールインデックス参照（またはマスキング）です。

ブールマスクの作成

配列に対して比較演算子（<, >, ==, !=, <=, >=）を適用すると、元の配列と同じ形状を持つ**ブール配列（boolean array）**が返されます。このブール配列の各要素は、元の配列の対応する要素が条件を満たすかどうか（True または False）を示します ``。

Python

```
data = np.arange(5) # [0 1 2 3 4]
mask = data < 3
print(data)
print(mask)
# 出力:
# [0 1 2 3 4]
#
```

複数の条件を組み合わせることも可能です。論理演算子 & (AND), | (OR), ~ (NOT) を使用します。Pythonの and, or, not ではなく、ビット演算子 &, |, ~ を使う点に注意してください。また、演算子の優先順位のため、複数の条件式は通常、括弧 () で囲む必要があります ``。

Python

```
data = np.arange(10)
# 2より大きく、かつ7より小さい要素に対するマスク
mask_combined = (data > 2) & (data < 7)
print(data)
print(mask_combined)
# 出力:
# [0 1 2 3 4 5 6 7 8 9]
#
```

マスクを使った要素の選択

作成したブール配列（マスク）を、そのまま角括弧 の中に指定することで、マスクが `True` である位置に対応する要素**のみ**を選択できます。結果として得られる配列は、通常、選択された要素を含む**1次元配列**になります。

Python

```
data = np.arange(5) # [0 1 2 3 4]
```

```
mask = data > 2     #

# マスクを使って要素を選択
selected_elements = data[mask]
print(selected_elements)
# 出力: [3 4]
```

条件の作成と選択を一行で書くことも一般的です ``。

```
Python


# 偶数の要素のみを選択
even_elements = data[data % 2 == 0]
print(even_elements)
# 出力: [0 2 4]
```

この方法は多次元配列にも適用できますが、結果は通常1次元配列になります ``。

```
Python


arr_2d = np.arange(6).reshape((2, 3))
# [[0 1 2]
#  [3 4 5]]

# 3より大きい要素を選択
selected_gt_3 = arr_2d[arr_2d > 3]
print(selected_gt_3)
# 出力: [4 5]
```

マスクを使った値の変更

ブールマスクは、代入演算子の左辺で使用することもできます。これにより、マスクが True である要素のみに値を代入（変更）できます ``。

```
Python


data = np.arange(5) # [0 1 2 3 4]
print(f"Before: {data}")

# 2未満の要素を99に変更
```

```
data[data < 2] = 99
print(f"After:  {data}")
# 出力:
# Before: [0 1 2 3 4]
# After:  [99 99  2  3  4]
```

ブールインデックス参照はコピーを作成する

基本的なスライシングがビューを返すことが多いのに対し、ブールインデックス参照によって要素を選択した場合、通常は元のデータのコピーが作成されます ``。

Python

```
original_arr = np.arange(5) # [0 1 2 3 4]
selected_copy = original_arr[original_arr > 2] # [3 4]

# 選択された配列 (コピー) を変更
selected_copy = 999

print(f"Selected (copy): {selected_copy}")
print(f"Original array (should be unchanged): {original_arr}") # 元の配列は変更されない
# 出力:
# Selected (copy): [999   4]
# Original array (should be unchanged): [0 1 2 3 4]
```

これは、ブールマスクによって選択される要素が、元の配列のメモリ上で必ずしも連続して配置されているとは限らないためです。NumPyは選択された要素を保持するために新しい配列を作成する必要があり、結果としてコピーが生成されます。この挙動は、基本的なスライシング（ビュー）とは対照的であり、メモリ使用量や、結果の配列を変更した場合の副作用に関して重要です。ただし、上記のようにブールマスクを使って元の配列に値を代入する場合 (data[data < 2] = 99) は、元の配列が直接変更されます。

7. 配列の変更：形状変更と結合

配列の構造を変更したり、複数の配列を組み合わせたりする操作も頻繁に行われます。

配列の形状変更 (np.reshape または ndarray.reshape)

reshapeは、配列のデータを変更せずに形状（次元数や各次元のサイズ）を変更する機能です。ただし、新しい形状は元の配列の総要素数 (size) と互換性がなければなりません（つまり、総要素数が同じである必要があります）。
np.reshape(array, new_shape) という関数形式と、array.reshape(new_shape) というメソッド形式の両方があります ``。

Python

```
a = np.arange(6) # [0 1 2 3 4 5], shape=(6,)
print(f"Original array (shape {a.shape}):\n{a}")

# 2行3列の配列に形状変更
b = a.reshape((2, 3))
print(f"Reshaped array (shape {b.shape}):\n{b}")
# 出力:
# Original array (shape (6,)):
# [0 1 2 3 4 5]
# Reshaped array (shape (2, 3)):
# [[0 1 2]
#  [3 4 5]]
```

新しい形状の総要素数が元の配列と一致しない場合、エラーが発生します ``。

Python

```
# a.reshape((2, 4)) # これはエラーになる (2*4=8!= 6)
```

reshapeのnew_shapeタプルの一つの次元に -1 を指定すると、その次元のサイズは他の次元のサイズと元の配列の総要素数から自動的に計算されます ``。これは非常に便利です。

Python

```
# 元の配列 a は size=6

# 3行 x N列 に reshape (-1 が列数を自動計算)
c = a.reshape((3, -1))
print(f"Reshaped to (3, -1) -> shape {c.shape}:\n{c}")
# 出力:
# Reshaped to (3, -1) -> shape (3, 2):
# [[0 1]
#  [2 3]
#  [4 5]]

# M行 x 2列 に reshape (-1 が行数を自動計算)
d = a.reshape((-1, 2))
```

```
print(f"Reshaped to (-1, 2) -> shape {d.shape}:\n{d}")
# 出力:
# Reshaped to (-1, 2) -> shape (3, 2):
# [[0 1]
#  [2 3]
#  [4 5]]
```

reshapeが返す配列がビューかコピーかは、状況によります ``。データがメモリ上で連続しており、新しい形状と互換性のある配置が可能であれば、多くの場合ビューが返されます（この場合、reshapeの結果を変更すると元の配列も変更されます）。しかし、元の配列が非連続的なスライス操作などの結果である場合、reshapeはデータを再配置するためにコピーを作成することがあります。この挙動は必ずしも自明ではないため、reshapeの結果もスライスと同様にビューである可能性を考慮し、もし独立したコピーが必要な場合は new_arr = old_arr.reshape(...).copy() のように明示的に .copy() を使用するのが安全なプラクティスです。

配列の結合 (np.concatenate, np.vstack, np.hstack)

複数の配列を一つにまとめるための関数も用意されています。

- np.concatenate((a1, a2,...), axis=0): 配列のシーケンス (a1, a2,...) を、指定された既存の軸 (axis) に沿って結合します。結合する配列は、結合軸以外の次元の形状が一致している必要があります。
    - 行方向に結合 (axis=0): 各配列の列数が一致している必要があります ``。
      Python
      ```
      a = np.array([, ]) # shape (2, 2)
      b = np.array([])        # shape (1, 2)

      # axis=0 で結合 (行を追加)
      cat_axis0 = np.concatenate((a, b), axis=0)
      print(f"Concatenate along axis 0 (shape {cat_axis0.shape}):\n{cat_axis0}")
      # 出力:
      # Concatenate along axis 0 (shape (3, 2)):
      # [[1 2]
      #  [3 4]
      #  [5 6]]
      ```

    - 列方向に結合 (axis=1): 各配列の行数が一致している必要があります ``。
      Python
      ```
      c = np.array([, ]) # shape (2, 1)

      # axis=1 で結合 (列を追加)
      cat_axis1 = np.concatenate((a, c), axis=1)
      print(f"Concatenate along axis 1 (shape {cat_axis1.shape}):\n{cat_axis1}")
      # 出力:
      # Concatenate along axis 1 (shape (2, 3)):
      ```

```
# [[1 2 7]
#  [3 4 8]]
```

- np.vstack((a1, a2,...)): (Vertical Stack) 配列を**垂直方向（行方向）**に積み重ねます。2次元配列の場合、np.concatenateでaxis=0を指定するのと同じ効果があります。この一般的なケースのためのより簡単な構文です ``。
  Python
  ```
  # vstack は concatenate(axis=0) と同等 (2Dの場合)
  vstack_result = np.vstack((a, b))
  print(f"vstack result (shape {vstack_result.shape}):\n{vstack_result}")
  # 出力: (cat_axis0 と同じ)
  # vstack result (shape (3, 2)):
  # [[1 2]
  #  [3 4]
  #  [5 6]]
  ```

- np.hstack((a1, a2,...)): (Horizontal Stack) 配列を**水平方向（列方向）**に積み重ねます。2次元配列の場合、np.concatenateでaxis=1を指定するのとほぼ同じ効果があります ``。
  Python
  ```
  # hstack は concatenate(axis=1) と同等 (2Dの場合)
  # ただし、1次元配列に対する挙動は異なる (単純に連結される)
  d = np.array() # 1次元配列 shape (2,)
  e = np.array()# 1次元配列 shape (2,)
  hstack_1d = np.hstack((d, e))
  print(f"hstack 1D result (shape {hstack_1d.shape}):\n{hstack_1d}")
  # 出力:
  # hstack 1D result (shape (4,)):
  # [ 7  8  9 10]

  # 2次元配列の場合
  hstack_2d = np.hstack((a, c)) # a:(2,2), c:(2,1)
  print(f"hstack 2D result (shape {hstack_2d.shape}):\n{hstack_2d}")
  # 出力: (cat_axis1 と同じ)
  # hstack 2D result (shape (2, 3)):
  # [[1 2 7]
  #  [3 4 8]]
  ```

配列を結合する際の最も重要な点は、結合軸以外の次元の形状が一致していることを確認することです。例えば、axis=0（行方向）で結合する場合、配列の列数（shape）がすべて同じでなければなりません。axis=1（列方向）で結合する場合は、行数（shape）が同じである必要があります。vstackとhstackは、それぞれaxis=0とaxis=1（主に2次元配列の場合）での結合を暗黙的に行う便利なショートカットですが、np.concatenateはaxisパラメータを明示的に指定することで、より一般的で、特に高次元配列を扱う際に柔軟な制御を提供します。どの関数を使うにせよ、結合する配列の形状がどの

ように整合する必要があるかを理解することが、正しい結果を得るために不可欠です。

まとめ：配列結合関数の比較

関数	主要な結合軸	説明	例（概念）
np.concatenate(..., axis=0)	0 (行方向)	指定した軸（行）に沿って配列を結合。他の軸のサイズは一致	[,] と [,] → [,,,]
np.concatenate(..., axis=1)	1 (列方向)	指定した軸（列）に沿って配列を結合。他の軸のサイズは一致	[] と [] → []
np.vstack(...)	0 (行方向)	配列を垂直（行方向）に積み重ねる (concatenate(axis=0))	[] と [] → [,]
np.hstack(...)	1 (列方向)	配列を水平（列方向）に積み重ねる (concatenate(axis=1))	[,] と [,] → [,]

この表は、配列結合の主要な関数を比較したものです。特にnp.concatenateと、その一般的なユースケースであるvstackおよびhstackの関係性を理解するのに役立ちます。各関数が主にどの軸に沿って操作を行うかを把握することで、適切な関数を選択し、形状の互換性要件を理解する助けとなります。

8. ブロードキャスティング：賢い形状の異なる配列間の演算

ブロードキャスティング（Broadcasting）は、NumPyが形状の異なる配列間で算術演算を行うことを可能にする強力なメカニズムです。これにより、明示的なループや配列の複製（タイリング）を行うことなく、簡潔で効率的なコードを記述できます。最初は少し戸惑うかもしれませんが、NumPyを使いこなす上で非常に重要な概念です。

ブロードキャスティングの概念

基本的な考え方は、形状が異なる配列間で演算を行う際に、NumPyが小さい方の配列を大きい方の配列に合わせて仮想的に「引き伸ばし」、要素ごとの演算が可能な互換性のある形状にする、というものです。この「引き伸ばし」は、実際にメモリ上でデータを複製するわけではないため、効率的です。

最も単純な例は、スカラー（0次元配列とみなせる）と配列の演算です。スカラー値は、配列の各要素に対して演算されるように、配列と同じ形状に「ブロードキャスト」されます ,。

Python

```
a = np.arange(4) # [0 1 2 3]
print(a + 5)    # スカラー 5 が配列 a の各要素に加算される
```

```
# 出力: [5 6 7 8]

b = np.ones((2, 2))
print(b * 3)    # スカラー 3 が 2x2 配列の各要素に乗算される
# 出力:
# [[3. 3.]
#  [3. 3.]]
```

ブロードキャスティングのルール

2つの配列 arr1 と arr2 が演算可能かどうか、そして結果の形状がどうなるかは、以下のルールに従って決定されます。ルールは、配列の形状（shape）を末尾の次元から比較していくことで適用されます。

1. ルール**1**: 2つの配列の次元数 (ndim) が異なる場合、次元数が少ない方の配列の形状の左側（先頭）に**1**が追加され、次元数が揃えられる。
    - 例: arr1 の形状が (2, 3)、arr2 の形状が (3,) の場合、arr2 の形状は (1, 3) として扱われる ``。
2. ルール**2**: 比較している次元において、どちらかの配列のサイズが1である場合、そのサイズ**1**の次元が、もう一方の配列のその次元のサイズに合わせて引き伸ばされる（タイリングされるように扱われる）。
    - 例: ルール1の後、arr1 が (2, 3)、arr2 が (1, 3) の場合、arr2 の最初の次元（サイズ1）が arr1 に合わせてサイズ2に引き伸ばされ、arr2 は (2, 3) として扱われる ``。
3. ルール**3**: 比較している次元において、サイズが異なり、かつどちらのサイズも**1**ではない場合、これらの配列はブロードキャスト不可能であり、エラーが発生する ``。
    - 例: arr1 が (2, 3)、arr2 が (2, 4) の場合、最後の次元でサイズが不一致（3 vs 4）であり、どちらも1ではないため、ブロードキャストできない。

これらのルールがすべての次元で満たされれば（エラーが発生しなければ）、2つの配列はブロードキャスト可能です。結果の配列の形状は、各次元における比較サイズの大きい方になります。

コード例

- **2**次元配列と**1**次元配列（行ベクトル）の加算: 形状 (2, 3) の行列と形状 (3,) の行ベクトルの加算を考えます ``。

  ```
  Python
  matrix = np.arange(6).reshape((2, 3)) # [[0 1 2], [3 4 5]]
  row_vector = np.array()   # shape (3,)

  print("Matrix:\n", matrix)
  print("Row Vector:", row_vector)

  result = matrix + row_vector # ブロードキャスティング発生
  print("Matrix + Row Vector:\n", result)
  # 出力:
  # Matrix:
  ```

```
# [[0 1 2]
# [3 4 5]]
# Row Vector: [10 20 30]
# Matrix + Row Vector:
# [[10 21 32]
# [13 24 35]]
```

この場合、以下の手順でブロードキャストが適用されます:

1. matrix (2, 3) と row_vector (3,) の次元数が異なるため、ルール1により row_vector は (1, 3) として扱われます。
2. 形状 (2, 3) と (1, 3) を比較します。
    - 最後の次元: 3 と 3 で一致。OK。
    - 最初の次元: 2 と 1 で異なる。ルール2により、サイズ1の方がサイズ2に合わせて引き伸ばされます。row_vector (1, 3) が (2, 3) として扱われます。
3. 結果として、matrix (2, 3) と、仮想的に (2, 3) に引き伸ばされた row_vector の間で要素ごとの加算が行われます。row_vector が matrix の各行に加算される形になります。

- **2**次元配列と列ベクトルの加算: 形状 (2, 3) の行列と形状 (2, 1) の列ベクトルの加算を考えます ``。列ベクトルを作成するには、明示的に形状を指定する必要があります（例：reshapeやnp.newaxisを使う）。

```
Python
matrix = np.arange(6).reshape((2, 3)) # [[0 1 2], [3 4 5]]
col_vector = np.array([, ]) # shape (2, 1)

print("Matrix:\n", matrix)
print("Column Vector:\n", col_vector)

result = matrix + col_vector # ブロードキャスティング発生
print("Matrix + Column Vector:\n", result)
# 出力:
# Matrix:
# [[0 1 2]
# [3 4 5]]
# Column Vector:
# [
# ]
# Matrix + Column Vector:
# [[100 101 102]
# [203 204 205]]
```

この場合：

1. matrix (2, 3) と col_vector (2, 1) の次元数は同じです。
2. 形状を比較します。
    - 最後の次元: 3 と 1 で異なる。ルール2により、サイズ1の方がサイズ3に合わせて

引き伸ばされます。col_vector (2, 1) が (2, 3) として扱われます。
    - 最初の次元: 2 と 2 で一致。OK。
3. 結果として、matrix (2, 3) と、仮想的に (2, 3) に引き伸ばされた col_vector の間で要素ごとの加算が行われます。col_vector が matrix の各列に加算される形になります。

意図の明確化と **np.newaxis**

ブロードキャスティングは非常に強力ですが、暗黙的な次元の追加（ルール1）や引き伸ばしに依存するため、意図がコードから読み取りにくい場合があります。特に、1次元配列を行ベクトルとして扱うか列ベクトルとして扱うかは、ブロードキャスティングの結果に大きく影響します。
このような場合に、np.newaxis（またはエイリアスとして None）を使って、配列に明示的にサイズ1の次元を追加することが推奨されます。これにより、ブロードキャスティングの意図が明確になります ``。

Python

```
a = np.array() # shape (3,)

# a を列ベクトル (3, 1) として扱う
col_a = a[:, np.newaxis] # または a[:, None]
print(f"Original shape: {a.shape}")
print(f"Column vector shape: {col_a.shape}")
print(f"Column vector:\n{col_a}")
# 出力:
# Original shape: (3,)
# Column vector shape: (3, 1)
# Column vector:
# [
#
#  ]

# a を行ベクトル (1, 3) として扱う
row_a = a[np.newaxis, :] # または a[None, :]
print(f"Row vector shape: {row_a.shape}")
print(f"Row vector:\n{row_a}")
# 出力:
# Row vector shape: (1, 3)
# Row vector:
# [[1 2 3]]
```

np.newaxis を使用することで、ブロードキャスティングのルール1（暗黙的な1の追加）に頼る代わりに、配列がどのように他の配列と整合されるべきかをコード上で明示的に示すことができ、エラーを減らし、コードの可読性を向上させることができます。

9. まとめ：NumPyの基礎をマスターする

本レポートでは、Pythonにおける数値計算の中核ライブラリであるNumPyの基本的な機能について解説しました。主要なトピックを振り返ってみましょう。

- **ndarray**: NumPyの中心となる、高速でメモリ効率の良い多次元配列オブジェクト。
- 配列の作成: Pythonリストからの作成 (np.array)、ゼロや1で埋められた配列の作成 (np.zeros, np.ones)、連番配列の作成 (np.arange, np.linspace) など、様々な方法。
- 配列の属性: 配列の特性を知るための属性 (ndim, shape, size, dtype)。特に shape と dtype の重要性。
- ベクトル化演算: ループを使わずに配列全体に高速な計算を行う機能（算術演算子、ufunc）。
- インデックス参照とスライシング: 配列の要素や部分配列にアクセスする方法。スライスは通常ビュー（元の配列への参照）を返す点に注意。
- ブールインデックス参照: 条件に基づいて要素を選択または変更する方法。こちらは通常コピーを返す。
- 形状変更と結合: 配列の形状を変える (reshape)、複数の配列を結合する (concatenate, vstack, hstack)。
- ブロードキャスティング: 形状の異なる配列間で演算を可能にする強力なルールセット。np.newaxisによる意図の明確化。

NumPyは、その計算効率と強力な配列操作機能により、Pythonのデータサイエンス、機械学習、科学技術計算のエコシステムにおいて不可欠な存在です。Pandas、SciPy、Matplotlib、Scikit-learnといった多くのライブラリがNumPyを基盤として利用しています。

ここで紹介した機能はNumPyの基礎ですが、これらを習得することで、より高度な数値計算やデータ分析への道が開かれます。さらに深く学ぶための次のステップとしては、以下のようなトピックが考えられます。

- より高度なインデックス参照（ファンシーインデックス参照）
- 線形代数モジュール (np.linalg)
- 乱数生成モジュール (np.random) の詳細
- ファイル入出力機能
- PandasやMatplotlibとの連携

NumPyの能力を最大限に引き出す鍵は、実際にコードを書き、様々な操作を試してみることです。ぜひ、ここで学んだ知識を基に、実践を重ねてみてください。