CH-7: Array-Oriented Programming with NumPy

Introduction

- **NumPy Overview:** Introduced in 2006, provides high-performance `ndarray` (or `array`). Operations are up to 100x faster than Python lists, essential for big-data applications.
- **Dependencies:** Over 450 Python libraries, including Pandas, SciPy, and Keras, rely on NumPy.
- **Key Features:** Enables *array-oriented programming* with concise functional-style data manipulation, reducing manual loops and bugs.
- **Lists vs Arrays:** Lists require nested loops for multidimensional processing, whereas NumPy arrays simplify such operations.
- **Intro to Pandas:** Offers flexible collections like `Series` (1D) and `DataFrame` (2D) for handling mixed data types, custom indexing, and missing/unstructured data.
- **Progression:** Covers lists, arrays, `Series`, and `DataFrames`, leading to tensors in deep learning chapters.

Creating arrays from Existing Data

- Importing NumPy:
  - Recommended to import as `np` for convenient access.

```
In [1]: import numpy as np
```

- Creating Arrays:
  - Use `np.array()` to create an array from a list or other collections.

```
In [3]: numbers = np.array([1,2,4,5,7,9,2,5])

In [4]: numbers
Out[4]: array([1, 2, 4, 5, 7, 9, 2, 5])
```

- Array Type:
  - Arrays created using `np.array()` are of type `numpy.ndarray`.

```
In [3]: type(numbers)
Out[3]: numpy.ndarray
```

Creating arrays from Existing Data

- Array Display:
  - Arrays are displayed with `array([...])`.
  - Values are separated by commas and aligned based on the largest value's field width.
- Multidimensional Arrays:
  - Create arrays from nested lists for multiple dimensions.

```
In [5]: marr = np.array([[1, 2, 3], [4, 5, 6]])

In [6]: marr
Out[6]:
array([[1, 2, 3],
       [4, 5, 6]])
```

  - Rows and columns are auto-aligned for readability.
- Formatting:
  - NumPy formats arrays based on dimensions and largest value width, ensuring uniform alignment.

array Attributes

- Array Creation:
  - Arrays can be created using `np.array()` from lists.

```
In [7]: integers = np.array([[1, 2, 3], [4, 5, 6]])

In [8]: floats = np.array([0.0, 0.1, 0.2, 0.3,
    0.4])

In [9]: integers
Out[9]:
array([[1, 2, 3],
       [4, 5, 6]])

In [10]: floats
Out[10]: array([0. , 0.1, 0.2, 0.3, 0.4])
```

- Data Types:
  - Use the `dtype` attribute to check an array's element type.

```
In [11]: integers.dtype
Out[11]: dtype('int64')

In [12]: floats.dtype
Out[12]: dtype('float64')
```

  - Common types: `int64`, `float64`, `bool`, `object`.

array Attributes

- Dimensions:
  - `ndim`: Number of dimensions.
  - `shape`: Tuple of dimensions (rows, columns).

```
In [16]: a = np.array([[1,2,3],[4,5,6]])

In [17]: a.ndim
Out[17]: 2

In [18]: a.shape
Out[18]: (2, 3)
```

- Size and Element Size:
  - `size`: Total number of elements.
  - `itemsize`: Number of bytes per element.

```
In [19]: a.size
Out[19]: 6

In [20]: a.itemsize
Out[20]: 8
```

array Attributes

- Iterating Through Arrays:
  - Iterate over rows and columns in a 2D array:

```
In [22]: for row in a:
    ...:     for col in row:
    ...:         print(col, end=' ')
    ...:     print()
    ...:
1 2 3
4 5 6
```

  - Use `flat` for one-dimensional iteration:

```
In [23]: for i in numbers.flat:
    ...:     print(i, end= ' ')
    ...:
1 2 4 5 7 9 2 5
```

- Key Observations:
  - `NumPy` auto-aligns array formatting for readability.
  - Default types are efficient for performance (e.g., `int64`, `float64`).
  - Arrays are iterable, supporting both external and internal iterations.

Filling arrays with Specific Values

- Functions for Array Creation:
  - zeros: Creates arrays filled with 0s.
  - ones: Creates arrays filled with 1s.
  - full: Creates arrays filled with a specified value.
  - **Default Behavior:** zeros and ones create arrays of float64 values by default.
- Specifying Dimensions:
  - Use an integer for 1D arrays:

    ```
    In [24]: np.zeros(5)
    Out[24]: array([0., 0., 0., 0., 0.])
    ```

  - Use a tuple of integers for multidimensional arrays:

    ```
    In [25]: np.zeros((2,5))
    Out[25]:
    array([[0., 0., 0., 0., 0.],
           [0., 0., 0., 0., 0.]])
    ```

  - Use the `dtype` keyword argument to set the element type (e.g., int, float).

Filling arrays with Specific Values

- Using full:
  - Create arrays filled with a specified value:

```
In [26]: np.full((3,2), 15)
Out[26]:
array([[15, 15],
       [15, 15],
       [15, 15]])
```

Creating arrays from Ranges

- Creating Integer Ranges with arange:
  - Similar to Python's built-in range, but optimized for arrays.
  - **Syntax:** `np.arange(start, stop, step)`

```
In [32]: np.arange(5)
Out[32]: array([0, 1, 2, 3, 4])

In [33]: np.arange(1,5)
Out[33]: array([1, 2, 3, 4])

In [34]: np.arange(0,10,2)
Out[34]: array([0, 2, 4, 6, 8])
```

- Creating Floating-Point Ranges with linspace:
  - Produces evenly spaced floating-point numbers, including the end value.
  - **Syntax:** `np.linspace(start, stop, num=num_of_pts)`

```
In [35]: np.linspace(1,3,4)
Out[35]: array([1.        , 1.66666667, 2.33333333,
    3.        ])
```

Creating arrays from Ranges

- Reshaping Arrays:
  - Use the reshape method to transform arrays into new dimensions.
  - The total number of elements must match the original array.

```
In [39]: np.arange(10).reshape(2,5)
Out[39]:
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

- Displaying Large Arrays:
  - For arrays with 1000+ elements, NumPy truncates the middle rows and columns in the output.

```
In [40]: np.arange(10000).reshape(10,1000)
Out[40]:
array([[   0,    1,    2, ...,  997,  998,  999],
       [1000, 1001, 1002, ..., 1997, 1998, 1999],
       [2000, 2001, 2002, ..., 2997, 2998, 2999],
       ...,
       [7000, 7001, 7002, ..., 7997, 7998, 7999],
       [8000, 8001, 8002, ..., 8997, 8998, 8999],
       [9000, 9001, 9002, ..., 9997, 9998, 9999]])
```

Creating arrays from Ranges

- Performance:
  - Use `arange` and `linspace` for better performance compared to Python loops.
  - Example: Time operations using `%timeit` in IPython for optimization.

List vs. array Performance: Introducing %timeit

- Key Differences in Performance:
  - **Speed Advantage of Arrays:** Arrays (using NumPy) execute compute-intensive tasks significantly faster than Python lists.
  - **Scalability:** With larger data sizes, arrays maintain better performance, often two orders of magnitude faster.
- %timeit Magic Command:
  - Purpose: Times the execution of a statement and reports:
    - Average execution time.
    - Standard deviation over multiple runs.
  - Defaults:
    - Executes a statement in a loop and runs the loop multiple times (default: 7 loops).
    - Chooses iteration counts automatically based on the statement's runtime.
  - Customization:
    - -n: Number of iterations per loop.
    - -r: Number of loops.

Array Operators

- Key Differences in Performance:
  - **Speed Advantage of Arrays:** Arrays (using NumPy) execute compute-intensive tasks significantly faster than Python lists.
  - **Scalability:** With larger data sizes, arrays maintain better performance, often two orders of magnitude faster.
- %timeit Magic Command:
  - Purpose: Times the execution of a statement and reports:
    * Average execution time.
    * Standard deviation over multiple runs.
  - Defaults:
    * Executes a statement in a loop and runs the loop multiple times (default: 7 loops).
    * Chooses iteration counts automatically based on the statement's runtime.
  - Customization:
    * -n: Number of iterations per loop.
    * -r: Number of loops.

Array Operators

- NumPy provides various operators for performing element-wise operations on arrays. These operations enable concise and efficient computations.
- Arithmetic Operations with Arrays and Scalars
  - Element-wise operations apply the operation to every element in the array.

```
In [5]: import numpy as np

In [6]: a = np.array(range(1,6))

In [7]: a*2
Out[7]: array([ 2,  4,  6,  8, 10])

In [8]: a**3
Out[8]: array([1, 8, 27, 64, 125], dtype=int32)
```

Array Operators

- Broadcasting: When one operand is a scalar, it behaves like an array with the same shape and all elements having the scalar value.

```
#Equivalent to
In [10]: a*[2,2,2,2,2]
Out[10]: array([ 2,  4,  6,  8, 10])
```

- Augmented Assignments: Augmented assignments modify the original array.

```
In [13]: a = np.array(range(1,6))

In [14]: a
Out[14]: array([1, 2, 3, 4, 5])

In [15]: a+=2

In [16]: a
Out[16]: array([3, 4, 5, 6, 7])
```

Array Operators

- Arithmetic Operations Between Arrays:
  - Arrays of the same shape can perform element-wise arithmetic.
  - Mixed types (integer and float) result in a floating-point array.

```
In [17]: a
Out[17]: array([3, 4, 5, 6, 7])
In [18]: b = np.linspace(1,4.75,5)
In [19]: b
Out[19]: array([1.    ,1.9375, 2.875, 3.8125, 4.75  ])
In [20]: a*b
Out[20]: array([ 3.    , 7.75  , 14.375, 22.875, 33.25 ])
```

- Comparing Arrays: Comparisons are element-wise, resulting in a Boolean array.

```
In [37]: a = np.array([1,3,2,9,4])
In [38]: b = np.array([2,4,1,6,8])
In [39]: a>=3
Out[39]: array([False,  True, False,  True,  True])
In [40]: a<b
Out[40]: array([ True,  True, False, False,  True])
```

NumPy Calculation Methods

- NumPy arrays provide various methods to perform calculations using their elements.
- By default, these methods operate on all elements, ignoring the array's shape.
- You can also perform calculations along specific dimensions using the axis keyword argument.
- Array-Wide Calculations
  - Methods such as sum, min, max, mean, std (standard deviation), and var (variance) operate on all array elements by default.

```
In [45]: grades.sum()
Out[45]: 1054

In [46]: grades.mean()
Out[46]: 87.83333333333333

In [47]: grades.std()
Out[47]: 8.792357792739987

In [48]: grades.var()
Out[48]: 77.30555555555556
```

NumPy Calculation Methods

```
In [49]: grades.min()
Out[49]: 70

In [50]: grades.max()
Out[50]: 100
```

- Calculations by Rows or Columns:
  - Using the `axis` argument, you can compute calculations for specific dimensions:
    - `axis=0`: Operates along columns (vertically).
    - `axis=1`: Operates along rows (horizontally).

```
In [57]: grades.sum(axis=0) # sum of each column
Out[57]: array([381, 341, 332])

In [58]: grades.min(axis=1) # min of each row
Out[58]: array([70, 87, 77, 81])
```

Question Use NumPy random-number generation to create an array of twelve random grades in the range 60 through 100, then reshape the result into a 3-by-4 array. Calculate the average of all the grades, the averages of the grades in each column and the averages of the grades in each row.

Universal Functions

- NumPy offers dozens of standalone universal functions that perform various element-wise operations.
- Each performs its task using one or two array or array-like (such as lists) arguments.
- Some of these functions are called when you use operators like + and * on arrays.
- Basic Universal Function Examples:
  - **Square Root:** Using `np.sqrt` to calculate the square root of each element.
  - **Addition:** Using `np.add` to add corresponding elements of two arrays.

```
In [59]: a = np.array([1,25,100])
In [60]: np.sqrt(a)
Out[60]: array([ 1.,  5., 10.])
In [61]: b = np.arange(1,4)*5
In [62]: np.add(a,b)
Out[62]: array([  6,  35, 115])
# Equivalent to a+b
```

Universal Functions

- Broadcasting with Universal Functions: Broadcasting allows operations between arrays of different shapes if their dimensions are compatible.
  - **Scalar Broadcasting:**

```
In [64]: np.multiply(a,2)
Out[64]: array([  2,   50, 200])
#Eqivalent to a*2
```

  - **Broadcasting with Arrays of Different Shapes:**

```
In [69]: a
Out[69]: array([  1,   25, 100])
In [70]: b
Out[70]: array([ 5, 10, 15, 20, 25, 30])
In [71]: b=b.reshape(2,3)
In [72]: np.multiply(a,b)
Out[72]:
array([[   5,  250, 1500],
       [  20,  625, 3000]])
#If the shapes of the arrays are incompatible
#for broadcasting, a ValueError occurs.
```

Universal Functions

- **Math:** `add`, `subtract`, `multiply`, `divide`, `sqrt`, `power`, etc.
- **Trigonometry:** `sin`, `cos`, `tan`, `arcsin`, `arccos`, etc.
- **Bit Manipulation:** `bitwise_and`, `bitwise_or`, `invert`, etc.
- **Comparison:** `greater`, `less`, `equal`, `logical_and`, etc.
- **Floating Point:** `floor`, `ceil`, `isnan`, `isinf`, etc.

→ For the full list, visit the NumPy Universal Functions Documentation

Indexing and Slicing

- Indexing One-Dimensional Arrays:
  - Use square brackets to access elements by their index (e.g., `array[0]` to get the first element).
- Indexing Two-Dimensional Arrays:
  - To select an element, use a tuple with row and column indices: `array[row, column]`
  - Example: `grades[0, 1]` selects the element in row 0, column 1.
- Selecting Rows in Two-Dimensional Arrays:
  - Select a single row by specifying one index: `grades[1]` selects the second row.
  - Select multiple rows using slice notation: `grades[0:2]` selects the first two rows.
  - To select non-sequential rows, use a list of indices: `grades[[1, 3]]`.

Indexing and Slicing

- Selecting Columns in Two-Dimensional Arrays:
  - Use a colon `:` to select all rows in a specific column: `grades[:, 0]` selects the first column.
  - To select consecutive columns, use slice notation: `grades[:, 1:3]`.
  - To select specific columns, use a list of indices: `grades[:, [0, 2]]`.

IPython Session

Question Create an array of the following form:

```
array([[10,  20,  30,   40,   50],
       [60,  70,  80,  90,   100],
       [110, 120, 130, 140, 150]]).
```

Then, write statements to perform following tasks:

- Select the second row.
- Select the first and third rows.
- Select the middle three columns.

Views: Shallow Copies

- A shallow copy, also called a view, shares the same data as the original array but creates a new array object.
- Modifications in either the original array or the view are reflected in both, as they share the same data.
- Created using:
  - The `view()` method.
  - Slicing the array.

```
In [6]: a
Out[6]: array([1, 2, 3, 4, 5])
In [7]: b = a.view()
In [8]: a[1]=10
In [9]: a
Out[9]: array([ 1, 10,  3,  4,  5])
In [10]: b
Out[10]: array([ 1, 10,  3,  4,  5])
In [11]: b[1]/=10
In [12]: a
Out[12]: array([1, 1, 3, 4, 5])
In [13]: b
Out[13]: array([1, 1, 3, 4, 5])
```

Views: Shallow Copies

- **Slicing example:**

```
In [19]: a
Out[19]: array([ 1, 20,  3,  4,  5])

In [20]: b=a[2:]

In [21]: b
Out[21]: array([3, 4, 5])

In [22]: b[1]*=10

In [23]: a
Out[23]: array([ 1, 20,  3, 40,  5])

In [24]: b
Out[24]: array([ 3, 40,  5])
```

Deep Copy

- A deep copy creates a completely independent copy of the original array, with its own data in memory.
- Modifications to the original array do not affect the deep copy, and vice versa.
- Created using the `copy()` method.

```
In [25]: a = np.arange(1,6)
In [26]: b=a.copy()
In [27]: b
Out[27]: array([1, 2, 3, 4, 5])
In [28]: a[2]*=10
In [29]: a
Out[29]: array([ 1,  2, 30,  4,  5])
In [30]: b
Out[30]: array([1, 2, 3, 4, 5])
In [31]: b[3]**=2
In [32]: b
Out[32]: array([ 1,  2,  3, 16,  5])
In [33]: a
Out[33]: array([ 1,  2, 30,  4,  5])
```

Reshaping and Transposing

- NumPy provides various other ways to reshape arrays.
- We've used array method reshape to produce two-dimensional arrays from one-dimensional ranges.
- reshape vs. resize:
  - The array methods `reshape` and `resize` both enable you to change an array's dimensions.
  - Method reshape returns a view (shallow copy) of the original array with the new dimensions. It does not modify the original array:

```
In [85]: grades = np.array([[90,80,70],[60,50,100]])
In [86]: grades
Out[86]:
array([[ 90,  80,  70],
       [ 60,  50, 100]])
In [87]: grades.reshape(1,6)
Out[87]: array([[ 90,  80,  70,  60,  50, 100]])
In [88]: grades
Out[88]:
array([[ 90,  80,  70],
       [ 60,  50, 100]])
```

Reshaping and Transposing

- Method resize modifies the original array's shape:

```
In [89]: grades.resize(1,6)

In [90]: grades
Out[90]: array([[ 90,  80,  70,  60,  50, 100]])
```

- flatten vs. ravel
  - You can take a multidimensional array and flatten it into a single dimension with the methods flatten and ravel.
  - Method flatten deep copies the original array's data:

```
In [92]: grades
Out[92]:
array([[ 90,  80,  70],
       [ 60,  50, 100]])
In [93]: flattened = grades.flatten()
In [94]: flattened
Out[94]: array([ 90,  80,  70,  60,  50, 100])
In [95]: grades
Out[95]:
array([[ 90,  80,  70],
       [ 60,  50, 100]])
In [96]: flattened[0] = 100
In [97]: flattened
Out[97]: array([100,  80,  70,  60,  50, 100])
In [98]: grades
Out[98]:
array([[ 90,  80,  70],
       [ 60,  50, 100]])
```

Reshaping and Transposing

- Method ravel produces a view of the original array, which shares the grades array's data:

```
In [99]: grades
Out[99]:
array([[ 90,  80,  70],
       [ 60,  50, 100]])

In [100]: raveled = grades.ravel()

In [101]: raveled
Out[101]: array([ 90,  80,  70,  60,  50, 100])

In [102]: grades
Out[102]:
array([[ 90,  80,  70],
       [ 60,  50, 100]])

In [103]: raveled[0]=100

In [104]: raveled
Out[104]: array([100,  80,  70,  60,  50, 100])

In [105]: grades
Out[105]:
array([[100,  80,  70],
       [ 60,  50, 100]])
```

Reshaping and Transposing

- Transposing Rows and Columns
  - You can quickly transpose an array's rows and columns—that is "flip" the array, so the rows become the columns and the columns become the rows.
  - The T attribute returns a transposed view (shallow copy) of the array.

```
In [106]: grades
Out[106]:
array([[100,  80,  70],
       [ 60,  50, 100]])

In [107]: grades.T
Out[107]:
array([[100,  60],
       [ 80,  50],
       [ 70, 100]])
```

Reshaping and Transposing

- Horizontal and Vertical Stacking
  - You can combine arrays by adding more columns or more rows—known as horizontal stacking and vertical stacking. the columns and the columns become the rows.
  - Horizontal Stacking (hstack): Adds columns to combine arrays.

```
In [108]: grades = np.array([[100, 96, 70], [100, 87, 90]])
     ...: grades2 = np.array([[94, 77, 90], [100, 81, 82]])

In [109]: combined = np.hstack((grades, grades2))

In [110]: combined
Out[110]:
array([[100,  96,  70,  94,  77,  90],
       [100,  87,  90, 100,  81,  82]])
```

  - Vertical Stacking (vstack): Adds rows to combine arrays.

```
In [111]: comb = np.vstack((grades,grades2))

In [112]: comb
Out[112]:
array([[100,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```