COMP 1023 Introduction to Python Programming

Exercises on NumPy

COMP 1023 Teaching Team

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Exercise 1

Write a function `analyze_array` that takes a list of numbers, converts it into a NumPy array, and returns a dictionary containing the array's shape, mean, standard deviation, and maximum value. Use the following header:

```
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
result = analyze_array(data)
print(result)
# Outputs:
# { 'shape': (10,), 'mean': 5.5, 'std': 2.8722813232690143, 'max': 10 }
```

Exercise 1 Solutions

```python
import numpy as np

def analyze_array(data):
    # Convert the list to a NumPy array
    arr = np.array(data)

    # Calculate the required statistics
    result = {
        'shape': arr.shape,
        'mean': np.mean(arr),
        'std': np.std(arr),
        'max': np.max(arr)
    }

    return result
```

Exercise 2

Write a function `replace_outliers` that takes a 1D NumPy array and two numbers: `threshold` and `replacement_value`. The function should replace all elements in the array whose absolute value is greater than the `threshold` with the `replacement_value`. Use NumPy's boolean array indexing, not a Python loop.

```
data = np.array([1, 3, -5, 10, -2, 8, -12])
threshold = 5
replacement_value = 0

result = replace_outliers(data, threshold, replacement_value)
print(result)
# Output:
# array([1, 3, -5, 0, -2, 0, 0])
```

Useful functions:

- `numpy.absolute()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.absolute.html`
- `numpy.ndarray.copy()`:
  `https://numpy.org/devdocs/reference/generated/numpy.ndarray.copy.html`

Exercise 2 Solutions

```python
import numpy as np

def replace_outliers(data, threshold, replacement_value):
    # Create a copy to avoid modifying the original array
    result = data.copy()

    # Use boolean indexing to find elements where absolute value > threshold
    # and replace them with replacement_value
    result[np.absolute(result) > threshold] = replacement_value

    return result
```

Exercise 3

Write a function `reshape_and_sum` that takes a 1D NumPy array and two integers `rows` and `cols`. The function should:

1. Reshape the array into a 2D array with the given `rows` and `cols`.
2. Return a 1D array where each element is the sum of the corresponding column in the reshaped matrix.

If the array cannot be reshaped to the specified dimensions (due to size mismatch), return `None`.

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

result = reshape_and_sum(data, rows, cols)
print(result)
# Output:
# array([15, 18, 21, 24])

# Reshaped matrix would be:
# [[ 1,  2,  3,  4],
#  [ 5,  6,  7,  8],
#  [ 9, 10, 11, 12]]
# Column sums: 1+5+9=15, 2+6+10=18, 3+7+11=21, 4+8+12=24
```

Useful attributes or functions:

- `numpy.ndarray.size`:
  `https://numpy.org/doc/stable/reference/generated/numpy.ndarray.size.html`
- `numpy.reshape`:
  `https://numpy.org/doc/stable/reference/generated/numpy.reshape.html`
- `numpy.sum()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.sum.html#numpy.sum`

Exercise 3 Solutions

```python
import numpy as np

def reshape_and_sum(data, rows, cols):
    if data.size != rows * cols:
        return None

    # Reshape and sum columns in one line
    return np.sum(data.reshape(rows, cols), axis=0)
```

Exercise 4

Write a function `create_border` that takes a 2D NumPy array and an integer `border_width`. The function should add a border of zeros around the array with the specified width. The output should be a new array with the original array in the center, surrounded by zeros.

```
data = np.array([[1, 2], [3, 4]])
border_width = 1

result = create_border(data, border_width)
print(result)
# Output:
# array([[0, 0, 0, 0],
#        [0, 1, 2, 0],
#        [0, 3, 4, 0],
#        [0, 0, 0, 0]])
```

Useful attributes or functions:

- `numpy.shape`:
  `https://numpy.org/doc/stable/reference/generated/numpy.ndarray.shape.html`
- `numpy.zeros()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.zeros.html`

Exercise 4 Solutions

```python
import numpy as np

def create_border(data, border_width):
    # Get the original dimensions
    original_rows, original_cols = data.shape

    # Calculate new dimensions with border
    new_rows = original_rows + 2 * border_width
    new_cols = original_cols + 2 * border_width

    # Create a new array of zeros with the expanded dimensions
    bordered_array = np.zeros((new_rows, new_cols), dtype=data.dtype)

    # Place the original array in the center
    bordered_array[border_width:border_width + original_rows,
                   border_width:border_width + original_cols] = data

    return bordered_array
```

Exercise 5

Write a function `handle_special_values` that takes a NumPy array and returns a modified version where:

1. All `NaN` (Not a Number) values are replaced with -1.
2. All infinite values (both positive and negative infinity) are replaced with the array's maximum finite value (ignoring infinites and NaNs)
3. If there are no finite values in the array, return an array of zeros with the same shape.

```
import numpy as np
# np.nan is a special floating-point value that stands for "Not a Number".
# np.inf is a constant that represents positive infinity.
# -np.inf is a constant that represents negative infinity.
data = np.array([1.0, np.nan, 3.0, np.inf, -np.inf, 5.0])

result = handle_special_values(data)
print(result)
# Output:
# array([1., -1., 3., 5., 5., 5.])
```

Useful attributes or functions:

- `numpy.ndarray.copy()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.ndarray.copy.html`
- `numpy.isfinite()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.isfinite.html`
- `numpy.isinf()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.isinf.html`
- `numpy.isnan()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.isnan.html`
- `numpy.max()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.max.html`
- `numpy.zeros_like()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.zeros_like.html`

Exercise 5 Solutions

```python
import numpy as np

def handle_special_values(data):
    result = data.copy()

    # Check for finite values
    finite_vals = result[np.isfinite(result)]

    if len(finite_vals) > 0:
        max_finite = np.max(finite_vals)
        # Replace infinities with max finite value
        result[np.isinf(result)] = max_finite
        # Replace NaNs with -1
        result[np.isnan(result)] = -1
        return result
    else:
        return np.zeros_like(data)
```

Exercise 6

Write a function called `moving_average` that takes two parameters

- `arr`: A 1D NumPy array of numbers.
- `window_size`: An integer representing the size of the moving average window.

The function should apply a moving average to `arr` and return a new 1D NumPy array containing the average values. For each position i from 0 to `arr.size - window_size`, the output should be mean of `arr[i:i+window_size]`. The output array should have a length of `arr.size - window_size + 1`.
Note: `arr.size` is the number of elements in the array `arr`.

You must use NumPy operations without using explicit Python for or while loops or list comprehensions.

```python
import numpy as np

# Example 1
arr1 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
window1 = 3
output1 = moving_average(arr1, window1)
print("Input 1:", arr1)
print("Window size:", window1)
print("Output 1:", output1)
# Expected Output 1: [2. 3. 4. 5. 6. 7. 8. 9.]
# Explanation: (1+2+3)/3 = 2, (2+3+4)/3=3, ... , (8+9+10)/3=9

# Example 2
arr2 = np.array([2, 4, 1, 5, 9, 2, 6, 5, 3, 5])
window2 = 4
output2 = moving_average(arr2, window2)
print("\nInput 2:", arr2)
print("Window size:", window2)
print("Output 2:", output2)
# Expected Output 2: [3.   4.75 4.25 5.5  5.5  4.25 4.75]
# Explanation: (2+4+1+5)/4 = 3, (4+1+5+9)/4=4.75, ... , (6+5+3+5)/4=4.75
```

Useful attributes or functions:

- `numpy.ndarray.size`:
  `https://numpy.org/doc/stable/reference/generated/numpy.ndarray.size.html`
- `numpy.arange()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.arange.html`
- `numpy.mean()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.mean.html`

Exercise 6 Solutions

```python
def moving_average(arr, window_size):
    # Create indices for all windows
    indices = np.arange(window_size) + np.arange(arr.size - window_size + 1)[:, None]

    # Use advanced indexing to create the window matrix
    windows = arr[indices]
    # Compute mean along the second axis (axis=1)
    return np.mean(windows, axis=1)
```

Exercise 7

Write a function `find_closest_points` that takes two 2D NumPy arrays `points_a` and `points_b`, where each row represents a point in n-dimensional space. The function should return two arrays:

- `closest_indices`: For each point in `points_a`, the index of the closest point in `points_b` (using Euclidean distance).
- `min_distances`: The corresponding minimum distances for each point in `points_a`.

Use broadcasting to compute all pairwise distances efficiently without explicit loops.

```python
import numpy as np

points_a = np.array([[1, 2], [4, 5], [7, 8]])
points_b = np.array([[2, 3], [5, 6], [8, 9], [0, 1]])

closest_indices, min_distances = find_closest_points(points_a, points_b)
print(closest_indices)
print(min_distances)
# Outputs:
# array([0, 1, 2])
# array([1.41421356, 1.41421356, 1.41421356])

# Point [1,2] in A is closest to point [2,3] in B (index 0), distance = 1.414
# Point [4,5] in A is closest to point [5,6] in B (index 1), distance = 1.414
# Point [7,8] in A is closest to point [8,9] in B (index 2), distance = 1.414
```

Useful attributes or functions:

- `numpy.newaxis`:
  `https://numpy.org/doc/2.2/reference/constants.html`
- `numpy.argmin()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.argmin.html`
- `numpy.min()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.min.html`
- `numpy.sqrt()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.sqrt.html`
- `numpy.sum()`:
  `https://numpy.org/doc/stable/reference/generated/numpy.sum.html`

Exercise 7 Solutions

```python
import numpy as np

def find_closest_points(points_a, points_b):
    # Compute pairwise squared Euclidean distances using broadcasting
    # Expand dimensions to enable broadcasting: (m, 1, n) - (1, k, n) = (m, k, n)
    diff = points_a[:, np.newaxis, :] - points_b[np.newaxis, :, :]

    # Sum along the last axis to get squared distances: (m, k)
    squared_distances = np.sum(diff**2, axis=2)

    # Find the minimum distance indices for each point in points_a
    closest_indices = np.argmin(squared_distances, axis=1)

    # Get the actual Euclidean distances (square root of squared distances)
    min_distances = np.sqrt(np.min(squared_distances, axis=1))

    return closest_indices, min_distances
```