Lecture 11: Introduction to NumPy

Morten Rieger Hannemose, Vedrana Andersen Dahl

Fall 2023

Today's lecture

1. Introduction to NumPy (ca. 5 min)
2. NumPy Arrays and Operations (ca. 30 min)
3. Previous exam question (ca. 15 min)

Motivation

- Lists can contain numbers, and we can perform any computation on them we desire.
  - Is this not enough?

Example: Adding Lists

```python
list1 = [1, 2, 3]
list2 = [4, 5, 6]
result = []
for i in range(len(list1)):
    result.append(list1[i] + list2[i])
```

Motivation

- Lists can contain numbers, and we can perform any computation on them we desire.
  - Is this not enough?
- For numerical data in arrays, lists are slower and less practical.
- NumPy provides
  - *n*-dimensional arrays
  - *tools* to work with these arrays.
- NumPy allows vectorized operations for efficient array calculations.
- Operations can be performed element-wise without explicit looping.

Example: Adding Lists

```python
list1 = [1, 2, 3]
list2 = [4, 5, 6]
result = []
for i in range(len(list1)):
    result.append(list1[i] + list2[i])
```

Example: Adding NumPy arrays

```python
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
result = arr1 + arr2
```

NumPy

```
import numpy as np
```

- A Widely used package in scientific computing, data analysis, and machine learning.
- It is the de facto standard for working with numerical data in Python.
- Several other libraries are built on top of NumPy, such as Pandas, SciPy, Scikit-learn, and Scikit-image.
- We use arrays to represent matrices and vectors.
- Don't call your files `numpy.py`

NumPy Arrays: Multidimensional Arrays

Working with 2D Arrays

```
arr = np.array([1,2,3])                         # 1D array
arr = np.array([[1, 2, 3], [4, 5, 6]]) # 2D array (2x3)
print("Arr has shape", arr.shape)
print(arr[0][1]) # looks like list, but inconvenient
print(arr[0, 1])
print(arr[:, 1])
# slicing
print(arr[:, 1:3])
```

- NumPy supports multidimensional arrays.
- Accessing elements using indices, similar to lists.
- Reshaping:
  - The method `.reshape()`.
  - The attribute . The shape is mutable.

Mutability of Arrays and Binary Indexing

- In NumPy, arrays are mutable, like lists.
- However, changes to a slice directly affect the original array.

Boolean Indexing and Mutability

```python
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
# Create a boolean mask
mask = arr > 2
arr[mask] = 10
print(arr)
arr2 = arr[mask]
arr2[-1] = -5
print(arr)
```

NumPy Arrays: Creation

- Lists are designed to be used with `.append()`.
- For NumPy we should pre-allocate arrays
- Preallocation:
  - Don't iteratively grow the size of an array.
  - Create the array with the correct size before a for-loop.

Creating NumPy Arrays

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5]) # array from lists (of lists etc.)
arr_zeros = np.zeros((3, 4))    # array with only 0
arr_ones = np.ones((2, 3))      # array with only 1
arr_range = np.arange(0, 10, 2) # like range
```

NumPy Operations: Universal Functions (ufuncs)

Universal Functions

```
arr = np.array([1, 2, 3])

sqrt_arr = np.sqrt(arr)
exp_arr = np.exp(arr)
sin_arr = np.sin(arr)
```

- Universal Functions (ufuncs) apply element-wise operations.
- For example:
  - `np.sqrt()`
  - `np.exp()`
  - `np.sin()`

NumPy Operations: Broadcasting

Broadcasting

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([10, 20, 30])

result = arr1 + arr2
```

- Broadcasting enables operations on arrays of different shapes and sizes.
- NumPy handles shape mismatches.
  - We can add a 1D array to a 2D array.

Matrix Operations in NumPy

Matrix Operations

```
mat1 = np.array([[1, 2, 3],
                 [3, 4, 5],
                 [6, 7, 8]])

vec1 = np.array([5, 3, 2])

mat1 = np.array([[1, 2, 3],
                 [3, 4, 5],
                 [6, 7, 8]])

mat1.dot(vec1) # matrix-vector multiplication
mat1.dot(mat2) # matrix-matrix multiplication
mat1.T # matrix transpose
```

- NumPy provides syntax for linear algebra with matrices.

Statistics

Statistics

```
data = np.array([1, 2, 3, 4, 5])
mean_value = data.mean()
std_dev = data.std()
median = np.median(data)
```

- NumPy provides functions for statistical calculations.
- axis keyword (e.g., `.std(1)`).

Final notes

- Some often used NumPy methods are accessible in multiple ways
  - `x.mean()` is the same as `np.mean(x)`
- The method will almost always exist on the `np` module.

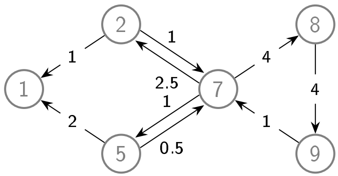
- There is a class called `numpy.matrix`
- Don't use it!
- From NumPy's documentation:
  - "It is no longer recommended to use this class, even for linear algebra. Instead use regular arrays. The class may be removed in the future."

Coding example

`node_divergence.py`, exam from June 2021.

Node divergence

A graph can be represented using a 2D array where every row contains a triplet of numbers (i, j, w_{ij}) representing one graph edge. Here, i is an index of from-node, j is an index of to-node, and w_{ij} is the weight of the edge from i to j. For example, consider the graph in the illustration and its representation using A.



$$A = \begin{bmatrix} 2 & 7 & 1 \\ 2 & 1 & 1 \\ 5 & 1 & 2 \\ 5 & 7 & 0.5 \\ 7 & 2 & 2.5 \\ 7 & 5 & 1 \\ 7 & 8 & 4 \\ 9 & 7 & 1 \\ 8 & 9 & 4 \end{bmatrix}$$

```
A = np.array([[2, 7, 1], [2, 1, 1], [5, 1, 2],
[5, 7, 0.5], [7, 2, 2.5], [7, 5, 1], [7, 8,
4], [9, 7, 1], [8, 9, 4]])
```

The divergence of node i is defined as

$$d_i = \sum_{\substack{j \\ \text{edge}ij}} w_{ij} - \sum_{\substack{j \\ \text{edge}ji}} w_{ji}.$$

So d_i is the difference between the sum of weights of all edges originating from i and the sum of weights of all edges ending in i. For example

$$d_7 = (2.5 + 1 + 4) - (1 + 0.5 + 1) = 5.$$

Problem definition

Create a function `node_divergence` that takes a 2D array representing a graph as input. The function should return an array containing sorted indices for graph nodes in one column and the divergence values for the corresponding nodes in the second column.

Node Divergence Solution

```python
import numpy as np

def node_divergence(A):
    nodes = np.unique(A[:, :2])
    return_arr = np.zeros((nodes.shape[0], 2))
    return_arr[:, 0] = nodes
    for i in range(nodes.shape[0]):
        node = nodes[i]
        divergence = A[A[:, 0] == node, 2].sum() - A[A[:, 1] == node, 2].sum()
        return_arr[i, 1] = divergence
    return return_arr
```