PROGRAMMING IN PYTHON I

Fast Numerical Computations in Python

Andreas Schörgenhumer

Institute for Machine Learning

JOHANNES KEPLER UNIVERSITY LINZ

Institute for Machine Learning

Copyright Statement

Contact

Andreas Schörgenhumer

Institute for Machine Learning
Johannes Kepler University
Altenberger Str. 69
A-4040 Linz

E-Mail: `schoergenhumer@ml.jku.at`

Write mails only for personal questions

Institute ML Homepage

Motivation

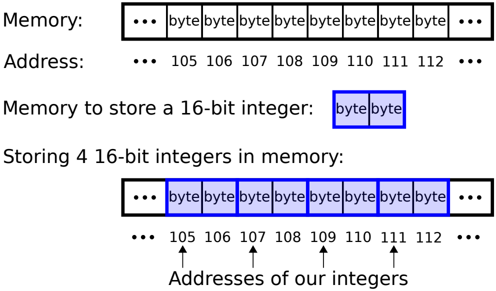
- Everything in Python is an object and code is executed line by line
  - Very convenient to use
  - Slow, since optimization of the code is difficult at runtime
- We can use modules in Python that allow us to write fast code in Python
  - By providing optimized functions (e.g., `NumPy`, ...)
  - By providing tools for optimizing Python-like code (e.g., `Numba`, `PyTorch`, `Tensorflow`, ...)

NumPy

- **NumPy** is the go-to module for numerical computations in Python
- Provides a large range of functionalities for performing scientific computations and handling array data
  - These functions are typically highly optimized and implemented in C
  - Access is still done via Python (you do not need to know C)
- NumPy mainly deals with (multidimensional) array data based on the `numpy.ndarray` object
- Documentation/Tutorials:
  https://numpy.org/doc/stable/index.html

Arrays in NumPy (1)

- Elements are stored as one block with contiguous addresses in memory
- Elements are fast to access since we can quickly compute their addresses



Arrays in NumPy (2)

- In Python, an element in a list is simply a reference to the corresponding Python object
  - Data types of objects are flexible
  - → Operations on elements are slower/clumsy (need to determine type of object before usage)
- In NumPy, an element in an array is (usually) a bit pattern that directly represents the stored value (and not a reference)
  - The array holds the information about the data type (encoding/decoding scheme for bits) used in array
  - Data type of elements in array is fixed (but we can create new arrays with a different data type)
  - **All elements** in an array have the **same data type**
  - → Operations on elements can be optimized and are faster

Multidimensional Arrays

- In Python, we already saw the concept of nested lists
  - Can be used to create 2D or nD arrays
  - Slow, since we have to access the sublists to access our elements
- In NumPy, we can store nD arrays as fast 1D arrays
  - Done by NumPy in the background
  - Store nD array in a **flat** manner
  - **Row-major** order: Consecutive elements of a row reside next to each other (NumPy default)
  - **Column-major** order: Consecutive elements of a column reside next to each other

Multidimensional Arrays: Example (1)

- We want to store a 2D array with 3 rows and 5 columns
  - 5 elements per row, 3 per column, 15 in total

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

- We can create a 1D array with 15 elements

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- We can say that
  - the first 5 (column) elements belong to the first row
  - the next 5 (column) elements belong to the second row
  - the last 5 (column) elements belong to the third row

  $\rightarrow$ row-major order

Multidimensional Arrays: Example (2)

- We agreed on row-major order
- Now, we want to access the element in the 4th column $c = 3$ and the 3rd row $r = 2$ (indices starting at 0 with $n_r = 5$ elements per row)

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

- We can compute the index in the 1D array via $n_r \cdot r + c = 5 \cdot 2 + 3 = 13$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- This is automatically done in the background for you, you do not have to worry about the correct index calculation

Indexing in NumPy

- Accessing NumPy arrays is similar to Python lists
  - Index via integers:
    `my_array[i]`
  - Slicing is possible and fast (since elements are consecutively stored in memory):
    `my_array[:i]`
- NumPy offers many more **fancy indexing** options
  - Indexing multi-dimensional arrays directly:
    `my_array[row, col]`
    `my_array[2, 4, 8, 5]`
  - Indexing using lists of indices, boolean index masks, ...
- More examples in the accompanying code file

Shapes and Axes in NumPy

- The **shape** defines the (multi-)dimensionality of an array
- Each dimension can be accessed using the corresponding **axis** (i.e., dimensions = axes)
- Example of a 2D array, i.e., an array with 2 axes

  ```
  [[1 2 3]
   [4 5 6]]
  ```

  The shape of this array is `(2, 3)`, i.e., the first axis has a length of 2 and the second axis has a length of 3
- Many NumPy methods in the provided library require to specify the axis on which to perform some operation