DSC 5101 – Computer Programming in DSAI

Part II - Working with Data

HO 04 - Numerical Computation using NumPy

Opim Salim Sitompul

Department of Data Science and Artificial Intelligence
Universitas Sumatera Utara

Co-funded by the Erasmus+ Programme of the European Union

Outline I

Introduction to NumPy

- This topic outlines techniques for effectively building, storing, and manipulating in-memory data in Python.
- Data can come from various types of datasets, including:
    - collections of documents,
    - collections of images,
    - collections of sound clips,
    - collections of numerical measurements, etc.

Introduction to NumPy

- All of the following representations obtained by transforming data into arrays of numbers are the first step in order to make the datasets analyzable.
  - Representing digital images as two-dimensional arrays of numbers containing pixel brightness across the area.
  - Representing sound-clips as one-dimensional array of intensity versus time.
  - Converting text into numerical representations, such as binary digits representing frequency of certain words or pairs of words.

Introduction to NumPy

- NumPy (*Numerical Python*) provides interface to store and operate on dense data buffers.
- NumPy arrays are like Python's built-in *list* type, but provides much more efficient storage and data operations as the size of arrays grow larger.
- NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python.

Introduction to NumPy

- To use NumPy in a program, we should import it and see the version by writing:

  ```
  import numpy
  numpy.__version__ → '1.19.5'
  ```

- It is also a common practice to import NumPy with an alias, such as:

  ```
  import numpy as np
  ```

Fixed-Type Array in Python

- Python offers option to store data as fixed-type data buffers.
- The built-in *array* module can be used to create dense arrays of a uniform type:

  import array
  L = list(range(10))
  A = array.array('i', L)
  A $\rightarrow$ **array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])**
- Type code *i* is indicating integer.

Creating Arrays from Python Lists

- To create arrays from Python lists:

  # integer array:

  np.array([1, 4, 2, 5, 3]) → **array([1, 4, 2, 5, 3])**
- NumPy is constrained to arrays that all contain the same type (and perform automatic *upcast* if necessary).

  np.array([1.5, 4, 2, 5, 3]) → **array([1.5, 4., 2., 5., 3.])**
- To explicitly set the data type of the resulting array, use *dtype* keyword.

  np.array([1, 4, 2, 5, 3], dtype='float32')

  → **array([1., 4., 2., 5., 3.], dtype=float32)**

Creating Arrays from Python Lists

- NumPy arrays can explicitly be multidimensional:

```
# Nested lists result in multidimensional array
np.array([range(i, i+3) for i in [2, 4, 6]]) →
```

array([[2, 3, 4],
[4, 5, 6],
[6, 7, 8]])

Creating Arrays from Scratch

- Some routines could be used to create arrays from scratch:
- Create a length 10 array of zeros

  np.zeros(10, dtype=int) $\rightarrow$

  array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
- Create a 3x5 floating-point array filled with 1s

  np.ones((3, 5), dtype=float) $\rightarrow$

  array([[1., 1., 1., 1., 1.],
  [1., 1., 1., 1., 1.],
  [1., 1., 1., 1., 1.]])
- Create a 3x5 floating-point array filled with 3.4

  np.full((3, 5), 3.14) $\rightarrow$

  array([[3.4, 3.4, 3.4, 3.4, 3.4],
  [3.4, 3.4, 3.4, 3.4, 3.4],
  [3.4, 3.4, 3.4, 3.4, 3.4]])

Creating Arrays from Scratch

- Creating Uniform Distribution Random Values Arrays:

  #Create a 3x3 array of uniformly distributed
  #random values between 0 and 1
  np.random.random((3, 3)) $\rightarrow$
  array([[0.28646093, 0.67326976, 0.540985],
  [0.23180937, 0.3483977 , 0.0354173],
  [0.32850528, 0.0014826 , 0.24844021]])

- Creating Normal Distribution Random Values Arrays:

  #Create a 3x3 array of normally distributed
  #random values with mean 0 and standard deviation 1
  np.random.normal(0, 1, (3, 3)) $\rightarrow$
  array([[0.17512094, 1.04104659, 1.09012586],
  [-1.08856692, 0.52793816, -0.79322176],
  [0.94165064, 0.7530818 , 0.88392039]])

Creating Arrays from Scratch

- Creating Random Integer Array

  #Create a 3x3 array of random integers in [0, 10]

  np.random.randint(0, 10, (3 , 3)) →

  **array([[7, 0, 2],
  [9, 6, 4],
  [3, 0, 1]])**

- Creating Identity Matrix

  np.eye(3) →

  **array([[1., 0., 0.],
  [0., 1., 0.],
  [0., 0., 1.]])**

- Creating Unitialized Array

  np.empty(3) →

  array([6.95291804e-310, 1.17106275e-311, -0.00000000e+000])

NumPy Standard Data Types

Table 1: NumPy arrays of a single type

Data Type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (*int64* or *int32*)
intc	int (*int64* or *int32*)
intp	Integer used for indexing (*int64* or *int32*)
int8	Integer (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned Integer (0 to 255)
uint16	Unsigned Integer (0 to 65535)
uint32	Unsigned Integer (0 to 4294967295)

NumPy Standard Data Types

Table 1: NumPy arrays of a single type (continued)

Data Type	Description
uint64	Unsigned Integer (0 to 18446744073709551615)
float_	Shorthand for *float64*
float16	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for *complex128*
complex64	Complex number: represented by two 32-bit floats
complex128	Complex number: represented by two 64-bit floats

NumPy Array Attributes

- Determining the base, size, shape, memory consumption, and data types of arrays.
- Example:

```
import numpy as np
np.random.seed(0)
# One-dimensional Array
x1 = np.random.randint(10, size=6)
# Two-dimensional Arrat
x2 = np.random.randint(10, size=(3, 4))
# Three-dimensional Array
x3 = np.random.randint(10, size=(3, 4, 5))
```

- The first parameter of *randint* shows the base of integer that will generate random number between 0 to 9.

NumPy Array Attributes

- Each array has attributes *ndim* (number of dimensions), *shape* (the size of each dimension), and *size* (total size of array).
- Displaying output of each attribute:
  print(x2.ndim) $\rightarrow$ **2**
  print(x2.shape) $\rightarrow$ **(3, 4)**
  print(x2.size) $\rightarrow$ **12**
  print(x2.dtype) $\rightarrow$ **int32**
  print(x2.itemsize, " bytes") $\rightarrow$ **4 bytes**
  print(x2.nbytes, " bytes") $\rightarrow$ **48 bytes**

Array Indexing

- In a one-dimensional array, accessing i^{th} value is performed using the index (counting from zero): x1 $\rightarrow$ **array([2, 5, 7, 7, 5, 0])**
  - x1[0] $\rightarrow$ **2**
  - x1[-1] $\rightarrow$ **0**
- In a two-dimensional array, item is accessed by comma-separated 2-tuple index:

```
x2 → array([[8, 3, 8, 3],
            [6, 5, 0, 2],
            [4, 1, 8, 9]])
x2[0, 0] → 8
```

Array Indexing

- In a three-dimensional array, item is accessed comma-separated 3-tuple index:

```
x3 → array([[[2, 6, 2, 6, 7],
             [5, 9, 7, 8, 5],
             [3, 1, 8, 5, 0],
             [4, 8, 6, 7, 6]],

            [[1, 8, 7, 5, 8],
             [0, 3, 6, 4, 9],
             [4, 4, 3, 5, 3],
             [0, 3, 5, 5, 5]],

            [[5, 5, 0, 7, 2],
             [1, 5, 3, 2, 5],
             [7, 7, 4, 6, 7],
             [7, 4, 5, 3, 9]]
x3[2, 3, 4] → 9
```

Array Slicing

References I

[1] VanderPlas J., Python Data Science Handbook: Essential Tools for Working with Data, O'Reilly, USA, 2016.

[2] Hunt, J., A Beginners Guide to Python 3 Programming, Springer Nature Switzerland AG, Switzerland, 2019.

[3] Hunt, J., Advance Guide to Python 3 Programming, Springer Nature Switzerland AG, Switzerland, 2019.