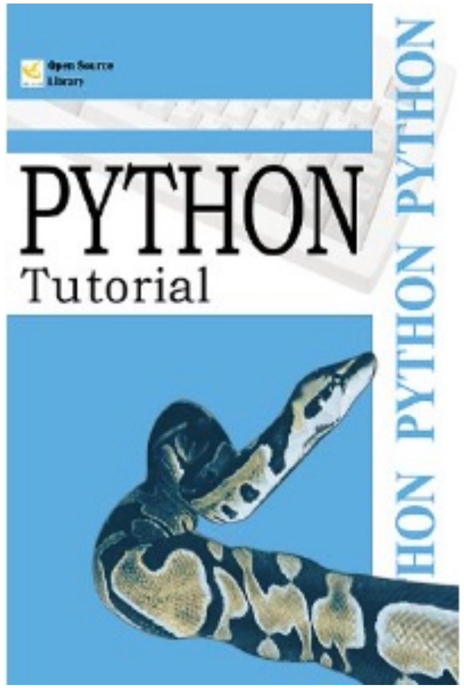
NumPy



Dra. Mª Dolores Rodríguez Moreno

Universidad de Alcalá
ISG

Objectives

Specific Objectives

- Learn how to use NumPy for numerical computing
- Understand the ndarray structure and its core attributes

Sources

- MIT OpenCourseWare: NumPy Book (pdf)
- Slideserve: Python Crash Course – NumPy Presentation
- Slides: Abhijeet Anand – Introduction to NumPy

Overview

- **Introduction**
- Why use NumPy?
- Creating NumPy Arrays
- The ndarray Data Structure
- Useful NumPy Functions
- 1D, 2D, 3D - arrays
- Exercise

Introduction …

- *Lists* are ok for storing small amounts of one-dimensional data
- But can't use directly with arithmetical operators (+, -, *, /, …) by element
- Need efficient arrays with arithmetic and better multidimensional tools
- NumPy –> *import numpy as np*
- Similar to lists, but much more capable — except with fixed size

Example: numpy – 1D

```
a = [2, 4, 6, 8, 10]

print(a[1:4])  # [4, 6, 8]

b = [[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]]

print(b[0]) # [2, 4, 6, 8, 10]

print(b[1][::2]) #[1, 5, 9]

x = [2, 4, 6, 8]

y = [1, 3, 5, 7]

z = x + y

print(z)  # [2, 4, 6, 8, 1, 3, 5, 7]

z = x - y #ERROR!!
```

Example: Operation to element level with lists

```
x = [2, 4, 6, 8]

y = [1, 3, 5, 7]

z = []

for i in range(len(x)):

    z.append(x[i] + y[i])

print(z)   # [3, 7, 11, 15]
```

...Introduction

- NumPy stands for Numerical Python
- It is a Python library used for working with arrays
- It is the fundamental package for scientific computing in Python
- It has :
  - Functions for working in the domain of linear algebra, Fourier transform, and matrices
  - Tools for integrating Fortran and C/C++ code
  - Random number generators
- It was created in 2005 by Travis Oliphant
- It is an open-source project and can be used freely

Overview

- Introduction
- **Why use NumPy?**
- Creating NumPy Arrays
- The ndarray Data Structure
- Useful NumPy Functions
- 1D, 2D, 3D - arrays
- Exercise

Why Use NumPy?

- In Python, we have lists that serve the purpose of arrays, but they are slow to process
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists
- Arrays are very frequently used in data science, where speed and resources are very important
- NumPy's array class is called *ndarray*

Overview

- Introduction
- Why use NumPy?
- **Creating NumPy Arrays**
- The ndarray Data Structure
- Useful NumPy Functions
- 1D, 2D, 3D - arrays
- Exercise

Creating NumPy Arrays

- NumPy arrays can be created in many ways:
  - Using np.array() to convert Python lists or tuples into arrays
  - With functions like np.arange(), np.linspace(), np.zeros(), np.ones()
  - Reading data from files (np.genfromtxt(), np.loadtxt())
  - Generating random values (np.random.random(), np.random.randint())

- Arrays can have any number of dimensions (1D, 2D, 3D...)
- The data type (*dtype*) can be specified during creation
- Initialization functions make large datasets fast and consistent

Initialization

All these **functions** create NumPy arrays (ndarray objects):

- np.array([1, 2, 3]): 1D array #You can specify type: np.array([1, 2, 3], dtype = float)
- np.array([[1, 2, 3], [4, 5, 6]]): 2D array
- np.arange(start, stop, step): Create an array (from start to stop with increments "steps"
- np.linspace(0, 2, 9): Add evenly spaced values between intervals to array of length
  [0.   0.25 0.5 0.75 1.   1.25 1.5 1.75 2.] # from 0 to 2, should be 9 values
- np.zeros((1, 2)): Create an array filled with zeros (1x2)
- np.ones((1, 2)): Create an array filled with ones
- np.random.random((5, 5)): Create a random array (5x5)
- np.empty((2, 2)): Create an empty array (2x2) of any value

Overview

- Introduction
- Why use NumPy?
- Creating NumPy Arrays
- **The ndarray Data Structure**
- Useful NumPy Functions
- 1D, 2D, 3D - arrays
- Exercise

The ndarray Data Structure

- NumPy introduces the class *ndarray*
- An N-dimensional, homogeneous collection of items
- Indexed using N integers (one per dimension)
- Defined by its shape and the data type of its elements

ndarray Attributes...

- ndim → number of dimensions (axes)
- shape → tuple describing the size in each dimension
- size → total number of elements
- dtype → element data type
- itemsize → memory size of one element in bytes
- data → memory buffer containing the actual elements

…ndarray Attributes

- The *data* attribute stores the raw memory buffer with the array's elements (direction in memory as a pointer in C)
- Usually, we don't access it directly — we use indexing or slicing to read and modify values
- Each item in an *ndarray* has the same type and size in memory
- Homogeneity ensures consistent interpretation of data

Some ndarray Methods

- tolist() ⟶ converts the array to a (nested) list
- copy() ⟶ returns a new independent copy of the array
- fill(value) ⟶ fills the array with a single scalar value
- sort() ⟶ sorts an array in place
- sort(axis=0): ⟶ sorts axis of array

Unlike np.sort(a), which returns a new sorted array, a.sort() modifies the array itself

Example: Methods

```
import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6]]) #Create the array

lst = a.tolist()

print(lst) # Output: [[1, 2, 3], [4, 5, 6]]

print(type(lst)) # Output: <class 'list’>

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

c = b.copy()        # creates a new array with its own memory

a = b

b[0] = 99

print(”a b c:", a, b, c)
```

Overview

- Introduction
- Why use NumPy?
- Creating NumPy Arrays
- The ndarray Data Structure
- **Useful NumPy Functions**
- 1D, 2D, 3D - arrays
- Exercise

Useful NumPy Functions

NumPy offers a wide range of functions beyond basic math operations, covering data generation, reshaping, statistics, and more

- abs() ⟶ absolute values
- cumsum(), min(), max() ⟶ cumulative and summary statistics
- randint(), shuffle(), transpose() ⟶ random generation and reshaping tools
- polyfit() ⟶ polynomial fitting for data analysis
- Many more: trigonometric (sin, cos), logical (logical_and), and exponential (exp, log)
- Also, the Python built-in len() returns the size of the first dimension
- Most NumPy functions are <u>vectorized</u>, meaning they operate on whole arrays at once

Example: General Functions

```
import numpy as np

a = np.array([-3, -1, 2, -4])

print(np.abs(a)) # Output: [3 1 2 4]

arr = np.array([1, 2, 3, 4])

print(np.cumsum(arr)) # Output: [ 1  3  6 10]   # running total

print(np.min(arr)) # Output: 1

print(np.max(arr)) # Output: 4
```

Vectorized Mathematical Operations (Universal Functions)

- np.add(x, y): Addition
- np.subtract(x, y): Subtraction
- np.divide(x, y): Division
- np.multiply(x, y): Multiplication
- np.sqrt(x): Square Root
- np.sin(x): Element-wise sine
- np.cos(x): Element-wise cosine
- np.log(x): Element-wise natural log
- np.dot(x, y): Dot producto (escalar product)
- np.roots([1, 0, -4]): Roots of given polynomial coefficients

Understanding Vectorization

- Vectorized means that operations are applied to every element of an array automatically — no “for” loops needed
- NumPy executes these operations internally in C, so they are extremely fast
- These are called vectorized operations, because they act on entire vectors or matrices at once

Example: uFuncs

```
import numpy as np

a = np.array([-3, -1, 2, -4])

print(np.abs(a)) # Output: [3 1 2 4]

b = np.array([1, 2, 3, 4])

c = np.array([5, 6, 7, 8])

print(np.add(b, c)) # Output: [ 6  8 10 12]

arr = np.array([1, 2, 3, 4])

print(np.cumsum(arr)) # Output: [ 1  3  6 10]   # running total

print(np.min(arr)) # Output: 1

print(np.max(arr)) # Output: 4
```

Remember, no LOOPS!!

Example: Operation to element level with lists

```
x = [2, 4, 6, 8]

y = [1, 3, 5, 7]

z = []

for i in range(len(x)):

    z.append(x[i] + y[i])

print(z)   # [3, 7, 11, 15]
```

Overview

- Introduction
- Why use NumPy?
- Creating NumPy Arrays
- The ndarray Data Structure
- Useful NumPy Functions
- **1D, 2D, 3D - arrays**
- Exercise

1-D Array

- An array that has 0-D arrays as its elements is called a uni-dimensional or 1-D array

- Elements: 7

Example: Numpy – 1D

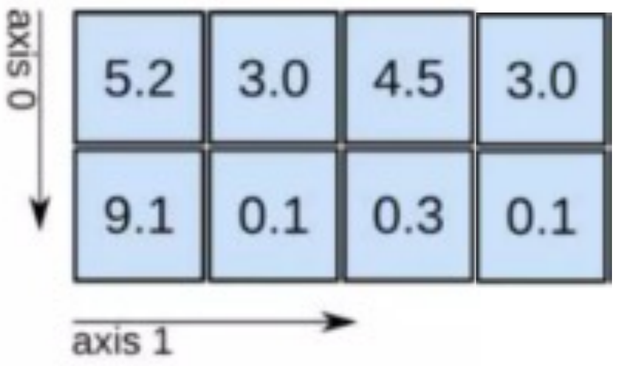
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

#Accessing Elements in 1D:

print(arr[0])  # Output: 1

#Negative Indexing:

print(arr[-1])  # Output: 5

#updating Elements

arr[1] = 10 # Output: [ 1 10  3  4  5]

#Appending Elements:

arr = np.append(arr, 6) # Output: [ 1 10  3  4  5  6]
```

Example: Numpy – 1D

```
#       arr = [ 1 10  3  4  5  6]
b = np.array([2, 2, 2, 2, 2, 2])

# Addition

print(arr + b) # Output: [ 3 12  5  6  7  8]

# Subtraction

print(arr - b) # Output: [-1  8  1  2  3  4]

# Multiplication

print("arr * b =", arr * b) # Output: [ 2 20  6  8 10 12]

# Division

print("arr / b =", arr / b) # Output: [0.5 5.  1.5 2.  2.5 3. ]
```

2-D Array

- An array that has 1-D arrays as its elements is called a 2-D array
- These are often used to represent a matrix or 2nd order tensors



- Dimension: 2 X 4

Example: numpy – 2D

```
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

#Accessing Elements in 2D:

print(arr_2d[1, 2])  # Output: 6

#Slicing:

print(arr_2d[0:2, 1:3])

#updating Elements

arr_2d[0, 0] = 10

#Appending Elements:

arr_2d = np.append(arr_2d, [[10, 11, 12]], axis=0)
```

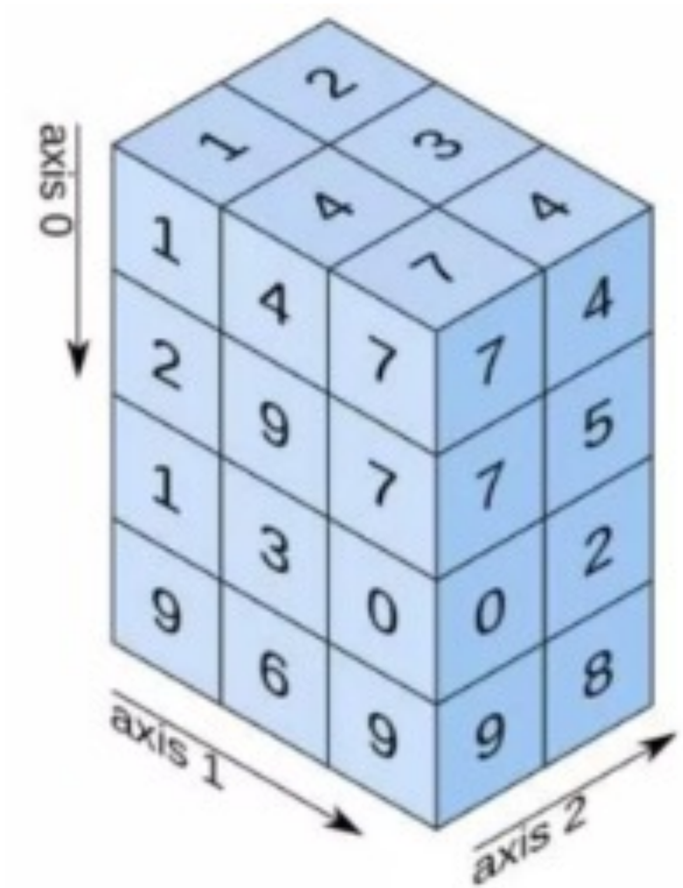
```
[[2 3]
 [5 6]]
```

```
[[10  2  3]
 [ 4  5  6]
 [ 7  8  9]]
```

```
[[10  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

3-D Arrays

- An array that has 2-D arrays (matrices) as its elements is called a 3-D array
- These are often used to represent a 3rd order tensor



Example: numpy – 3D

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

#Accessing Elements

print(arr[1, 0, 1])  # Output: 8

#Slicing

print(arr[:, 1, :])

#Updating Elements:

arr[0, 1, 1] = 20

#Appending Elements

arr = np.append(arr, [[[13, 14, 15], [16, 17, 18]]], axis=0)
```

```
[[ 4  5  6]
 [10 11 12]]
```

```
[[[ 1  2  3]
  [ 4 20  6]]
 [[ 7  8  9]
  [10 11 12]]
 [[13 14 15]
  [16 17 18]]]
```

Example: numpy – 3D

```
#Inserting Elements

arr = np.insert(arr, 1, [[[-1, -2, -3], [-4, -5, -6]]], axis=0)
```

```
[[[  1   2   3]
  [  4  20   6]]
 [[ -1  -2  -3]
  [ -4  -5  -6]]
 [[  7   8   9]
  [ 10  11  12]]
 [[ 13  14  15]
  [ 16  17  18]]]
```

```
#Deleting Elements

arr = np.delete(arr, 1, axis=0)
```

```
[[[ 1  2  3]
  [ 4 20  6]]
 [[ 7  8  9]
  [10 11 12]]
 [[13 14 15]
  [16 17 18]]]
```

Overview

- Introduction
- Why use NumPy?
- Creating NumPy Arrays
- The ndarray Data Structure
- Useful NumPy Functions
- 1D, 2D, 3D - arrays
- **Exercise**

Exercise

- Represent the grades of 3 students in 3 subjects
  1. Create a 2D Array to represent the grades
  2. Retrieve the grade of the second student in the third subject
  3. Modify the grade of the first student in the first subject
  4. Add a new student with different grades
  5. Insert a new subject at the second position
  6. Remove the second student
  7. Change the shape of the array to represent 2 students with 4 subjects each