Data Analysis with Python

Winter Semester 2018/2019
Lukas Arnold and Max Böhler

NumPy is the fundamental package for scientific computing with Python when it comes to large, multidimensional arrays (vectors) and matrices.

In addition to the fast computation of *NumPy* arrays (way more faster than Python Lists), the main benefit lies in the included mathematical operations like shape manipulation, sorting, selecting, I/O, basic linear algebra, basic statistical operations, random simulation and so on. One should hold in mind that *NumPy* arrays are not as flexible as Python Lists. While the latter one could store different data types in one list, each *NumPy* array only stores values of the same data type!

Examples and more detailed instructions how to use *numpy* can be found here: [http://www.numpy.org/ (http://www.numpy.org/)](http://www.numpy.org/)

Note: If numpy is not yet installed on your system, open the Anaconda prompt (or terminal on Unix systems) and type:

```
conda install numpy
```

In [ ]:
```
# This small Script shows how fast a NumPy Array in comparison to a Python L
ist is.
# Execute this script by pushing Shift+Enter

import time
import numpy as np
import sys

size_of_vec = 1000000

def pure_python_version():
    t1 = time.time()
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = [X[i] + Y[i] for i in range(len(X)) ]
    return time.time() - t1, Z

def numpy_version():
    t1 = time.time()
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y
    return time.time() - t1, Z


t1,py_list = pure_python_version()
t2,np_array = numpy_version()

list_mem = sys.getsizeof(py_list)/1000
array_mem = sys.getsizeof(np_array)/1000

print("Computational time for Python List: {} s and the object (memory) size
: {} kB".format(t1,list_mem))
print("Computational time for Numpy Array: {} s and the object (memory) size
: {} kB".format(t2,array_mem))

print("\nTherefore the numpy array is {:.4f} times faster!".format(t1/t2))
```

Array/Matrix indexing

Single element indexing for a 1-D works exactly like indexing for Python-Lists. It is 0-based, and accepts negative indices for indexing from the end of the array.

In [ ]:
```
# Example

# Different routines for the creation of NumPy 1-D arrays:
array_1 = np.array([1.0, 2.0, 3.0]) # Simple 1D array
array_2 = np.zeros(10) # Create an array with 10 elements with value 0
array_3 = np.arange(20.0) # Create an array from 0 to 19
array_4 = np.linspace(0,20,11) # Creates a evenly spaced array from 0 to 20
with 11 elements

print(array_1[-1]) # Print the last element of array_1
print(array_3) # Print all elements of array_3
print(array_4[array_4>10]) # Print all elements of array_4 which are bigger
than 10
```

2-D array indexing works in a similar way.

In [ ]:
```
# Example

array_2D = np.array([[1, 2, 3], [4, 5, 6]]) # create a 2D array (= 2x3 matri
x)

print(array_2D[1,2]) # print element in row 1 column 2
```

It is also possible to slice (accessing more than one element) numpy arrays.

In [ ]:
```
# Example

# Create a more complex Matrix

# First initalize Matrix
x = np.zeros(shape=(10,10)) # create a 10x10 zero-matrix

# Get the shape (number of rows/columns)
rows = x.shape[0]
columns = x.shape[1]

counter = 1

# Loop through each row/column
for i in range(0,rows):
    for j in range(0,columns):
        x[i,j] = counter
        counter = counter + 1

#Array Slicing
print(x[0,:]) # Print the first row of matrix x (using : as symbol for all e
lements in this row)
print(x[0,5:]) # Print the first row starting from column 5
print(x[:,3]) # Print all elements of column 3
```

numpy.loadtxt

The NumPy routine *loadtxt* is a convenient way to load data from a ASCII formatted file into a numpy array. In combination with *matplotlib* and array slicing it is also a fast way to plot data.

In [ ]:
```
# example
# For this example, it is necassary to download the 'simple_data.csv' file f
rom moodle!

data = np.loadtxt("simple_data.csv", delimiter=',')
#print(data)

x_values = data[:,0] # Column 0 contains all x-values
y_values = data[:,1] # Column 1 contains all y-values

import matplotlib.pyplot as plt

plt.plot(x_values, y_values)
plt.show()
```

Task 1: Load and manipulate data from a .csv file

1. Use NumPy's .loadtxt() routine to access the data stored in bitmap_data.csv and store it in a numpy array
2. Get the shape of this array as seen above
3. Loop through each row and column and check if each element is bigger or smaller than 150
   - if the element is bigger or equal 150 set the element value to 1
   - if the element is smaller than 150 set the value to 0

4. Visualize the modified data using the following Code snippet: (Don't forget to import matplotlib!)

```
plt.imshow(<Name of modified NumPy array>, cmap='binary_r')
plt.show()
```

What do you see?

In [ ]:

```
# Solution

# load the bitmap data
data = np.loadtxt("bitmap_data.csv", delimiter=',')

# show raw data
plt.imshow(data, cmap='binary_r')
plt.show()

# classic way
rows = data.shape[0]
columns = data.shape[1]

for i in range(0,rows):
    for j in range(0,columns):
        if data[i,j] >= 150:
            data[i,j] = 1
        else:
            data[i,j] = 0

plt.imshow(data, cmap='binary_r')
plt.show()

# numpy way
# relaod data
data = np.loadtxt("bitmap_data.csv", delimiter=',')

mask = data >= 150

data[~mask] = 0
data[mask] = 1

plt.imshow(data, cmap='binary_r')
plt.show()
```

Data manipulation using NumPy routines

Generate the derivation of a given set of data

The NumPy routine

```
np.gradient(y)
```

generates the derivation of a given function (data).

Smoothing via Moving-Average

Moving average is a simple operation used usually to suppress noise of a signal: we set the value of each point to the average of the values in its neighborhood. With NumPy this is done using the

```
np.convolve(y, np.ones((N,))/N), mode="same")
```

routine.
The derivation and mathematics of this routine is described under the following link:
http://matlabtricks.com/post-11/moving-average-by-convolution (http://matlabtricks.com/post-11/moving-average-by-convolution)

In [ ]:
```
# Example Derivation
import random

x = np.linspace(0,20,100) # Generate an array with 100 elements from 0 to 20
y = np.sin(x) # Calculate f(x) = sin(x)
y2 = np.cos(x) # Calculate f2(x) = cos(x)

dy = np.gradient(y,x) # Generate the derivation

plt.plot(x,y) # f(x)
plt.plot(x,y2) # f2(x)
plt.plot(x,dy,':') # df(x) which should be the same as f2(x)
plt.show()

# Example Smoothing

# Add noise to sin(x) using random numbers between -0.3 and 0.3
y_noise = y
for i in range(0,len(y_noise)):
    y_noise[i] = y_noise[i] + random.uniform(-0.3,0.3)


sm = np.convolve(y_noise, np.ones((5,))/5, mode='same') # Moving Average

plt.plot(x,y_noise)
plt.plot(x,sm)
plt.show()
```

Root finding

Root finding in combination with a derivative is one way to find the turning points of a function. A root exists if the function value assumes the value 0 at any point. Since NumPy arrays do not have continuous values (there is always a step between two elements) it is possible that there is no exact 0 value. Therefore one has to check if there is a sign change between two elements. If a sign change occured the root is between those two elements.

Task 2: RootFinding Alogorithm

1. Develop a function that implements the task described above. Proceed as follows:
   - The function should take two parameters x,y which are both two numpy arrays
   - Create an empty list in which the results will be stored
   - Loop through the y array and check if y[i] > 0 and y[i+1] < 0 or y[i] < 0 and y[i+1] > 0
   - If the condition is true use x[i] and x[i+1] two interpolate the value inbetween
   - Return the result list

In [ ]:

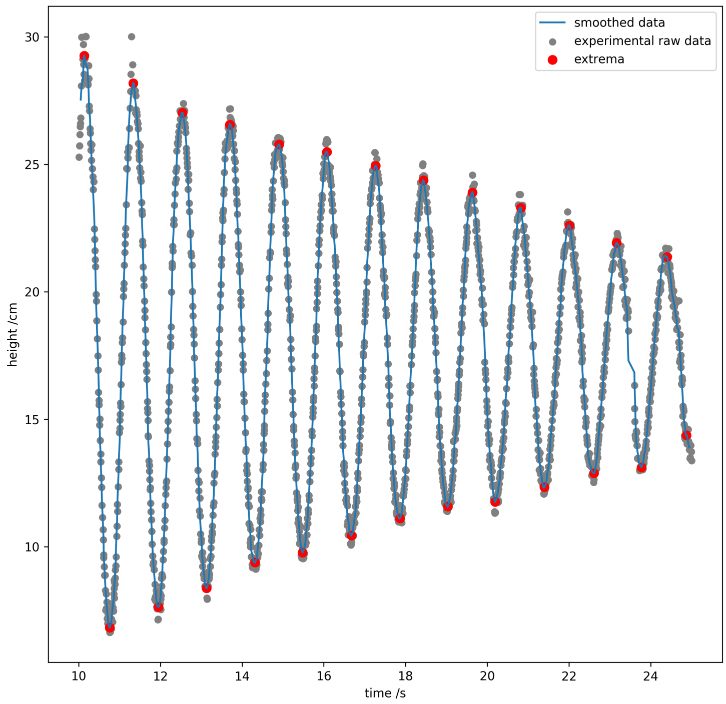
```
# Solution

def root_finding(x,y):
    root_list = []
    for i in range(0,len(y)-1):
        if y[i] == 0:
            root_list.append(x[i])
        elif ((y[i] > 0 and y[i+1] < 0) or (y[i] < 0 and y[i+1] > 0)):
            mid = (x[i] + x[i+1])/2
            root_list.append(mid)
    return root_list
```

In [ ]:

```
# Check if your function works:

x = np.linspace(0,20,1000)
y = np.sin(x)

roots = root_finding(x,y)
roots_y = np.zeros_like(roots)

print(roots)

plt.plot(x, y)
plt.scatter(roots, roots_y, color='red')
```

Task 3: Data Analysis using NumPy

1. Use NumPy's .loadtxt() routine to access the data stored in abstand.dat and store it in a numpy array
2. Plot column 0 against column 1 of the raw data
3. Define a mask to focus the data only on the first oszilations (e.g. t in [10 s, 25 s]) and mask out values larger than 50
4. Use NumPy's *.convolve* routine to smooth the data
5. Find the turning points (minima/maxima) of the masked data. Therefore compute the derivative in combination with the root_finding algorithm.
6. Finally try to create the following plot:



In [ ]:
```
# Solution

raw_data = np.loadtxt("distance_oszilator.dat", delimiter=' ')

# create mask and set filter conditions
mask = np.ones(len(raw_data), dtype=bool)
mask[np.where(raw_data[:,1] > 50)] = False
mask[np.where((raw_data[:,0] > 25) | (raw_data[:,0] < 10))] = False

# for convenience, define a new variable with the masked data
data=raw_data[mask, :]

x = data[:,0]
y = data[:,1]

plt.plot(x,y,':')

# N defines the widht of the smooting window
N = 11
sm = np.convolve(y, np.ones((N,))/N, mode='valid')
# the x values of the smoothed values are reduced by N points
N2 = N//2
sm_x = x[N2:-N2]
plt.plot(sm_x,sm)

plt.show()
```

In [ ]:
```
# compute the derivative
derivative = np.gradient(sm, sm_x)
plt.plot(sm_x, derivative)
# show the y=0 vicinity
plt.ylim(-2, 2)
```

In [ ]:
```
# find and print roots
raw_roots = root_finding(sm_x, derivative)
print(len(raw_roots), raw_roots)
```

In [ ]:
```
# create a list of roots, which are separated by at least dx
dx = 0.2

roots = []
for i in range(len(raw_roots)-1):
    if raw_roots[i+1] - raw_roots[i] > dx:
        roots.append(raw_roots[i])
roots.append(raw_roots[-1])
print(len(roots), roots)
```

In [ ]:
```
# plot the distance of the roots
distance_roots = []
for i in range(len(roots)-1):
    distance_roots.append(roots[i+1]-roots[i])
plt.plot(distance_roots)
```

In [ ]:
```
# interpolate the y-values of the extrema, i.e. roots of the derivative
roots_y = np.interp(roots, sm_x, sm)
plt.plot(x,y)
plt.plot(sm_x, sm)
plt.scatter(roots, roots_y, color='red', s=50)
```

In [ ]:

```
fig = plt.figure(figsize=(10,10))
plt.scatter(x,y, color='gray', s=25, label='experimental raw data')
plt.plot(sm_x, sm, label='smoothed data')
plt.scatter(roots, roots_y, color='red', s=50, label='extrema')
plt.legend()
plt.xlabel('time /s')
plt.ylabel('height /cm')
plt.savefig('final_plot.png', dpi=300, bbox_inches='tight')
plt.show()
```

In [ ]:

```
```