Stephanie Kaes <kaes@vision.rwth-aachen.de>
George Lydakis <lydakis@vision.rwth-aachen.de>

Exercise 0: Python Tutorial for Beginners

due **before** 2023-11-06

Important information regarding the exercises:

- The exercises are not mandatory. Still, we strongly encourage you to solve them! All submissions will be corrected. If you submit your solution, please read on:
- Use the Moodle system to submit your solution. You will also find your corrections there.
- Due to the large number of participants, we require you to submit your solution to Moodle **in groups of 3 to 4 students**. You can use the **Discussion Forum** on Moodle to organize groups.
- If applicable submit your code solution as a zip/tar.gz file named `mn1_mn2_mn3.{zip/tar.gz}` with your **matriculation numbers** (`mn`).
- Please do **not** include the data files in your submission!
- Please upload your pen & paper problems as PDF. Alternatively, you can also take pictures (.png or .jpeg) of your hand written solutions. Please make sure your handwriting is legible, the pictures are not blurred and taken under appropriate lighting conditions. All non-readable submissions will be discarded immediately.

Question 1: ..... $(\Sigma = 0)$

Broadcasting is a powerful feature in Python, particularly in libraries like NumPy, that allows you to perform operations on arrays of different shapes without explicitly having to reshape them. The basic rules for broadcasting are as follows:

1. If two arrays have a different number of dimensions, the shape of the smaller-dimensional array is padded with ones on the left side.
2. The sizes of the dimensions are then compared element-wise. Two dimensions are considered compatible when:
   - They are equal, or
   - One of them is 1.

Let's say you have two NumPy arrays, A and B, with the following shapes:

$$A : (3, 4)$$
$$B : (4,)$$

Because the arrays have different dimensions, broadcasting comes into play. The smaller array, B, is broadcasted to match the shape of the larger array, A, as follows:

$$B : (4,) \rightarrow (1, 4) \text{ (padding with ones)}$$

You can perform element-wise operations like addition or multiplication on these arrays without explicitly reshaping B. The broadcasting rules apply to operations on arrays with different shapes and are not limited to addition.

$$\text{result} = A + B \text{ (B is automatically broadcasted to match A's shape)}$$

Open the file python_tutorial02.py. Again, fill out the missing code pieces and document your results. Code for reading in the variables B, C, D, A, B, C, L already exists.

1. Define variable A as floating point matrix.

$$\begin{bmatrix} 0. & 8. & 0. & 6. & 6. \\ 0. & 5. & 6. & 0.8 & \\ 5. & 4. & 3. & 8. & 1. \\ 4. & 7. & 1. & 2. & 1. \\ 9. & 6. & 4. & 1. & 2. \end{bmatrix}$$

2. How can we print datatype and content of all of these objects?
3. Implement A+C and A-C.
4. Point out the differences between A*C, A@C, A/C and A@NUMPY.LINALG.INV(C).
5. Try to calculate C@A, C@B, B@C. What happens there?
6. Calculate B-D.
7. Calculate C+A. Why does this (not) work?
8. Please run the following code and explain why the variable A2 changed its value.

```
A2 = A
A[0,0] = 42
print(A2)
```

9. When applying *slicing*, we can make use of abbreviations. Instead of "0:n:1", we can apply ":n". Additionally, negative values can be used to grab array entries, counting from the last value in the array. Print matrix A...
   (a) ... without the first two columns.
   (b) ... but only the last column.
   (c) ... and show every second row and column.
   (d) ... with all rows in inverted order.
10. Create an array which includes array C twice. Use the function NP.APPEND. Explain how the AXIS argument is used.

Question 2: ..... $(\Sigma = 0)$

Open the file python_tutorial02.py and complement the missing code pieces. Sections in which you should write your own code are marked with TODO. In this first task we aim to invert a grayscale picture.

$$img = 255 - img$$

The image is saved as numpy array. Try out different versions of editing the picture and document the different runtimes.

1. Explain how a for-loop looks like in Python. Are there other types of loops, too?
2. Implement two nested loops (on rows and columns) and NEW = 255 - IMG[ROW, COL].

3. Use matrix operations 255 - IMG.
4. By defining a lookup table (np.array) with numbers 255, 254, ..., 0. Apply the table by using the OpenCV function CV.LUT.

Question 3: ..... $(\mathbf{\Sigma = 0})$

The objective of this task is to help you understand how to compute the inverse and pseudoinverse of a matrix using Python and NumPy.

1. Remember your math class. Which types of matrices are invertible?
2. Create a 2×2 matrix, A, with non-zero elements.
3. Compute the inverse of matrix A and store it in a variable.
4. Compute the pseudoinverse of matrix A and store it in a variable.
5. Compare the results of (2) and (3). Explain the concept of matrix inversion and pseudoinversion and the differences between the two.
6. Try this task with a matrix that is not invertible (singular), and observe what happens when computing the inverse and pseudoinverse.

Question 4: ..... $(\mathbf{\Sigma = 0})$

Matrix factorization is a technique used in linear algebra to break down a matrix into simpler, more interpretable components. Two common types of matrix factorization in NumPy are Singular Value Decomposition (SVD) and QR decomposition.

Singular Value Decomposition (SVD) SVD is a matrix factorization technique that decomposes a matrix A into three separate matrices - U, Σ (Sigma), and V^T (the transpose of V). The formula for SVD is:

$$A = U \cdot \Sigma \cdot V^T$$

Here's how you can perform SVD on matrix A using NumPy:

$$U, \Sigma, V^T = \text{np.linalg.svd}(A)$$

U, Σ, and V^T are representing the left singular vectors, singular values, and right singular vectors of the matrix A, respectively.

2. QR Decomposition QR decomposition factorizes a matrix A into two matrices - Q (orthogonal) and R (upper triangular). The formula for QR decomposition is:

$$A = QR$$

Here's how you can perform QR decomposition using NumPy:

$$Q, R = \text{np.linalg.qr}(A)$$

Q is an orthogonal matrix, and R is an upper triangular matrix.

These matrix factorization techniques are commonly used in various applications, including solving linear systems of equations, dimensionality reduction, and data compression. NumPy provides efficient implementations for these factorizations, making it a valuable tool for numerical computing and data analysis in Python.

1. Create a NumPy array representing a square matrix A with known coefficients. In this example, we'll use the following 3x3 matrix A:

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 1 & 0 & 1 \\ 3 & 2 & 3 \end{bmatrix}$$

2. Create a NumPy array representing a vector b with the same number of rows as the matrix A. In this example, we'll use the following vector b:

$$b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

3. Use NumPy to perform QR decomposition on the matrix A. You can use the `np.linalg.qr` function for this.
4. Solve the Linear Equation System:
   - Given the system $Ax = b$, use the QR decomposition results to find the solution vector x.
   - You can use the Q and R matrices to simplify the equation to $Rx = Q^T b$.
   - Solve for x using this simplified equation and NumPy.
5. Check the Solution:
   - Use the solution vector x to verify that it satisfies the original equation $Ax = b$.
   - Print the values of Ax and compare them to b.
6. Optional: Repeat for Different Matrices and Vectors:
   - If you want to practice further, repeat the above steps for different random matrices A and vectors b to solve multiple linear equation systems.

Question 5: ..... $(\mathbf{\Sigma = 0})$

The objective of this task is to practice function fitting in Python using libraries like NumPy and Matplotlib. You will be required to fit a mathematical function to a set of data points and create a plot to visualize the fit.

Generate a set of data points that follows a mathematical function with some noise. For this task, let's consider the function:

$$y = 3x^2 - 2x + 1 + \epsilon$$

Hint: You can use NumPy to create the 'x' values, and you may want to use 'np.random.normal' to add noise to the data.

(b) Your task is to fit a quadratic function to the generated data. The function you want to fit should be of the form:

$$f(x) = ax^2 + bx + c$$

Your goal is to find the values of a, b, and c that best fit the data.
Hint: You can use the 'curve_fit' function from 'scipy.optimize' to fit the function to the data.

(c) Create a plot that shows the original data points (with noise) and the fitted quadratic function. Additionally, label the axes and provide a legend to distinguish between the original data and the fitted function.
Hint: You can use Matplotlib to create the plot. Be sure to use 'plt.scatter' for the data points and 'plt.plot' for the fitted function. Label axes with 'plt.xlabel' and 'plt.ylabel', and create a legend with 'plt.legend'.

Import the necessary libraries (NumPy, Matplotlib, and ‘curve_fit‘ from ‘scipy.optimize‘).

Hint: You can import these libraries using ‘import numpy as np‘, ‘import matplotlib.pyplot as plt‘, and ‘from scipy.optimize import curve_fit‘.

2. Generate data with noise.

   Hint: Use ‘x = np.linspace(0, 10, 50)‘ to generate ‘x‘ values and ‘np.random.normal(0, 2, len(x))‘ to add noise to ‘y‘.

3. Define the quadratic function to fit.

   Hint: Define a Python function that takes ‘x‘, ‘a‘, ‘b‘, and ‘c‘ as parameters and returns the quadratic function.

4. Fit the function to the data using ‘curve_fit‘.

   Hint: Use ‘curve_fit‘ with your defined function and the data (‘x‘ and ‘y‘).

5. Extract the fitted parameters (a, b, c). Hint: The fitted parameters will be returned by ‘curve_fit‘.

6. Create the fitted function and plot the data and the fitted function. Hint: You can create ‘fitted_y‘ using your defined function with the fitted parameters, and then use Matplotlib for plotting as described in the task.