Numpy

Solve the following exercises and upload your solutions to Moodle until the specified due date.

Important Information!

Please try to *exactly match the output* given in the examples (naturally, the input can be different). We are running automated tests to aid in the correction and grading process, and deviations from the specified output lead to a significant organizational overhead, which we cannot handle in the majority of the cases due to the high number of submissions.

Make sure to use the *exact filenames* that are specified for each individual exercise.

Also, use the provided unit tests to check your scripts before submission (see the slides Handing in Assignments on Moodle). Feel free to copy the example text from the assignment sheet, and then change it according to the exercise task to match the output as best as possible.

In this assignment, it is of *particular importance* to wrap the printing that you see in the example outputs in `if __name__=='__main__':`, as your exercises essentially only consist of a function definition. Example - let's say the task is to write a function that doubles the float value that is passed:

```
def double(var: float) -> float:
    return var*2

if __name__ == '__main__':
    print(double(3.4))
```

Unless explicitly stated otherwise, you can assume correct user input and correct arguments.

You are *not allowed* to use any concepts and modules that have not yet been presented in the lecture.

Exercise 1 – Submission: `a10_ex1.py` 25 Points

Write a function `matrix_operations(size: int) -> tuple[np.nparray, np.nparray, int, tuple [int, int, int], tuple[int, int, int], int, np.ndarray]` that performs several matrix operations on a randomly generated 3D matrix of a given size.

The function performs the following steps:

- Check that `length` is > 0.
- A 3D matrix of shape $(size, size, size)$, randomly filled with integers between 0 and 100, is created.
- A separate sum across the first, the first two, and all dimensions of the matrix is computed.
- The index of the highest and the index of the lowest values are computed. These should be 3-dimensional indices for the matrix, not the flattened indices!
- The sum of all values on the border of the matrix is computed.
- The matrix is flattened into one dimension and normalised. The result should be a 1D array of size $3 \cdot size$ with values between 0 and 1.

The function should have no print output and should return one tuple containing the sums, the indices of the highest and lowest value, the border sums, and the flattened and normalised matrix.

Hints:

- To compute the on the borders of the matrix, think about it as a cube. The result should be the sum of all visible items (marked in green):

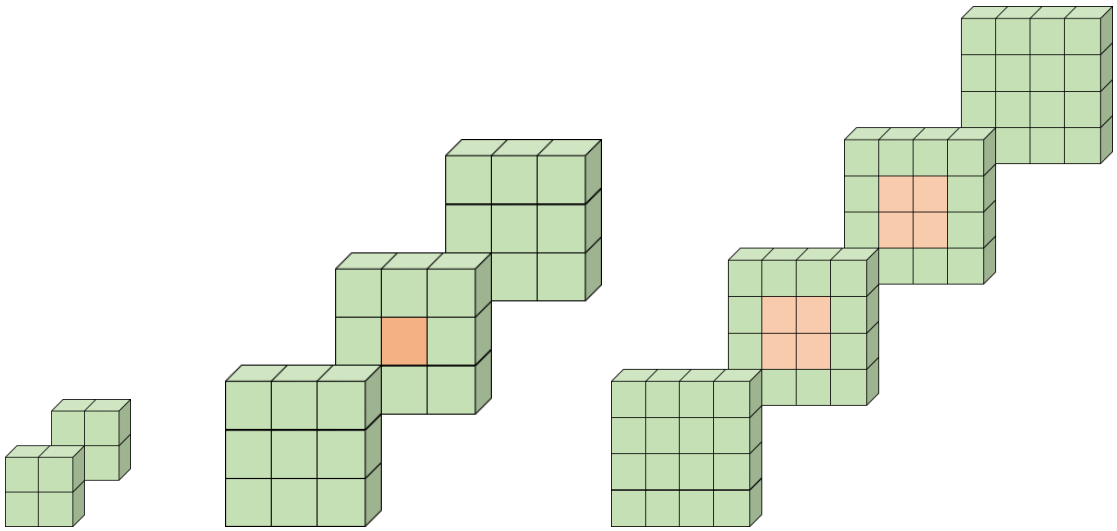


Figure 1: Green values should be added; red values should be ignored.

- `np.unravel_index` is useful to transform the index in a flattened array to the multi-dimensional index.

Example program execution:

```
np.random.seed(0)
axis_0_sum, axis_01_sum, axis_all_sum, max_index, min_index,  border_sum,
    normed_array = matrix_operations(3)
print("Sum of elements along axis 0: \n", axis_0_sum)
```

```
print("Sum of elements along axis 0 and 1: \n", axis_01_sum)
print("Sum of all elements: \n", axis_all_sum)
print("Index of maximum value: \n", max_index)
print("Index of minimum value: \n", min_index)
print("Sum of border values: \n", border_sum)
print("Normalized array: \n", normed_array)
```

Example output:

```
Sum of elements along axis 0:
 [[177 205 233]
 [192 104 144]
 [220  69 143]]
Sum of elements along axis 0 and 1:
 [589 378 520]
Sum of all elements:
 1487
Index of maximum value:
 (1, 0, 2)
Index of minimum value:
 (0, 1, 2)
Sum of border values:
 1475
Normalized array:
 [0.44303797 0.48101266 0.69620253 0.73417722 0.73417722 0.
 0.93670886 0.15189873 0.34177215 0.98734177 0.7721519  1.
 1.         0.03797468 0.62025316 0.70886076 0.37974684 0.98734177
 0.46835443 1.         0.91139241 0.35443038 0.20253165 0.86075949
 0.79746835 0.         0.13924051]
```

Example program execution:

```
np.random.seed(0)
axis_0_sum, axis_01_sum, axis_all_sum, max_index, min_index,  border_sum,
    normed_array = matrix_operations(4)
print("Sum of elements along axis 0: \n", axis_0_sum)
print("Sum of elements along axis 0 and 1: \n", axis_01_sum)
print("Sum of all elements: \n", axis_all_sum)
print("Index of maximum value: \n", max_index)
print("Index of minimum value: \n", min_index)
print("Sum of border values: \n", border_sum)
print("Normalized array: \n", normed_array)
```

Example output:

```
Sum of elements along axis 0:
 [[188 268 273 259]
 [205  99 127 112]
 [183 181 160 215]
 [231 202 140 245]]
Sum of elements along axis 0 and 1:
 [807 750 700 831]
Sum of all elements:
 3088
Index of maximum value:
 (2, 0, 1)
```

```
Index of minimum value:
 (3, 1, 2)
Sum of border values:
 2862
Normalized array:
 [0.44444444 0.47474747 0.64646465 0.67676768 0.67676768 0.09090909
 0.83838384 0.21212121 0.36363636 0.87878788 0.70707071 0.88888889
 0.88888889 0.12121212 0.58585859 0.65656566 0.39393939 0.87878788
 0.46464646 0.88888889 0.81818182 0.37373737 0.25252525 0.77777778
 0.72727273 0.09090909 0.2020202  0.80808081 0.6969697  0.7979798
 0.47474747 0.64646465 0.82828283 1.         0.88888889 0.49494949
 0.29292929 0.19191919 0.19191919 0.14141414 0.39393939 0.32323232
 0.65656566 0.09090909 0.57575758 0.32323232 0.31313131 0.74747475
 0.23232323 0.35353535 0.75757576 0.55555556 0.28282828 0.34343434
 0.         0.         0.36363636 0.53535354 0.05050505 0.38383838
 0.17171717 0.7979798  0.04040404 0.42424242]
```

Exercise 2 – Submission: `a10_ex2.py` 25 Points

Write a function `elements_wise(arr: np.ndarray, f)` that applies the given function `f` for each element in the given numpy array `arr` that has any number of dimensions. It transforms the input array `arr` in place by updating the results of `f` for each element directly into `arr`, i.e. no copy of the array is generated. Assume that the data type of elements in `arr` is compatible with the function `f`.

Example program execution:

```
def func(x):
    return x*x + 3*x + 2

a1 = np.array(range(2 * 2 * 3), dtype=float).reshape(2, 2, -1)
a2 = np.array(range(2 * 3), dtype=float).reshape(2, -1)
elements_wise(a1, func)
elements_wise(a2, func)
print(f"a1:\n {a1}")
print(f"a2:\n {a2}")
```

Example output:

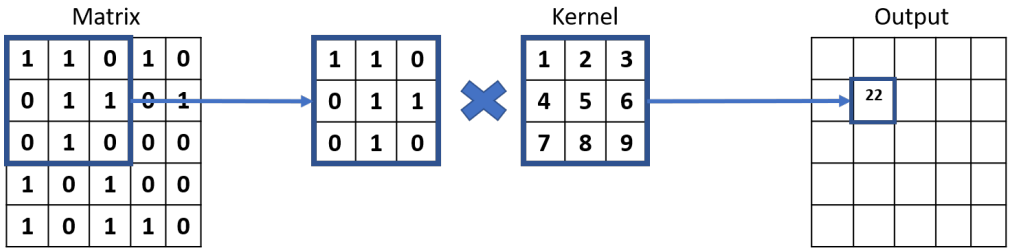
```
a1:
[[[  2.   6.  12.]
  [ 20.  30.  42.]]

 [[ 56.  72.  90.]
  [110. 132. 156.]]]

a2:
[[ 2.  6. 12.]
 [20. 30. 42.]]
```

Exercise 3 – Submission: `a10_ex3.py` 25 Points

Write a function `convolution(matrix: np.ndarray, kernel:np.ndarray, padding: int = 0) -> np.ndarray:` that performs a 2D convolution with a given kernel over a matrix. The kernel is a small 2D matrix that is slid over the input matrix and provides factors for a weighted sum of a local area of the matrix:



The function should ensure that the matrix and the kernel are 2-dimensional. In addition, the kernel has to be square and has to have an odd number of dimensions, and the matrix must be larger than the kernel.

The following is required:

- The input matrix should not get changed.
- The output matrix should be of the same shape as the input matrix. To ensure that, `padding` has to be be added on the border of the matrix. The padding should be enough to ensure that the kernel can slide over the matrix.
- The padding value is set with `padding`. It should default to 0.

Example program execution:

```
np.random.seed(0)
matrix = np.random.randint(0, 255, (32,32))
kernel = np.ones((3,3)) * -1
kernel[1,1] = 9

print(convolution(matrix, kernel))
```

Example output:

```
[[ 1178  -432   166 ...   -58  -710  1006]
 [  397   933  1236 ...  1415   929   447]
 [ 1055  -825  -790 ...  -329  -128   626]
 ...
 [  -59   378 -1121 ...  1503  -750   866]
 [  142   864  -549 ...  -561  -256  1498]
 [ -309  1507   688 ...  1671   308  -220]]
```

Hints:

- It is easier to pad the matrix in advance than to compute the padding on the fly.

Exercise 4 – Submission: `a10_ex4.py` 25 Points

Write a function `one_hot_encoding(arr: np.ndarray) -> np.ndarray:`
that performs one-hot-encoding for each element in the input 1D numpy array `arr`, and the returned result is a numpy 2D array in which i^{th} row is the encoding of the i^{th} element in `arr`. Note that before creating the encoding for each element, the unique values should be sorted. If `arr` is not 1D, a `ValueError(f"The function can work for 1D matrices, not <dim>D")` must be raised where `<dim>` is the dimension of the input array.

Example program execution:

```
a = np.array(["a", "a", "b", "c"])
print(one_hot_encoding(a))
a = np.array([10, 5, 15, 20])
print(one_hot_encoding(a))

a = np.array([[1, 2], [3, 4]])
try:
    print(one_hot_encoding(a))
except ValueError as e:
    print(f"ValueError: {e}")
```

Example output:

```
[[1. 0. 0.]
 [1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

[[0. 1. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]

ValueError: The function can work for 1D matrices, not 2D
```