NumPy

Solve the following exercises and upload your solutions to Moodle until the specified due date. Make sure to use the *exact filenames* that are specified for each individual exercise. 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.

Important Information!

We are running automated tests to aid in the correction and grading process, and deviations from the expected outputs lead to a significant organizational overhead, which we cannot handle in the majority of the cases due to the high number of submissions.

1. Please try to *exactly match the output* given in the examples (naturally, the input can be different). Feel free to copy the output text from the assignment sheet, and then change it according to the exercise task.

For example, if the exercise has an output of
`Number of cables: XYZ`
(where `XYZ` is some user input), do not write
`The number of cables: XYZ`
(additional `The` and lowercase `n`) or
`Number of cables:XYZ`
(missing space after the colon).

2. Furthermore, please don't have any lines of code that will be automatically executed when importing your module (except for what is asked by the exercise) as this will break our automated tests. Always execute your module before submitting to verify this !

For example, if you have some code to test your program and reproduce the example outputs, either comment/remove these lines or move them to the `"if __name__ == "main":"` section.

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

Write a function `extend(arr: np.ndarray, rows: int, cols: int, fill=None) -> np.ndarray` that extends a 2D numpy array `arr` to a given size (`rows, cols`) using the fill value determined by `fill` and returns this new array. The original `arr` must not be changed. The function works as follows:

- If `arr` is not 2D, a `ValueError(f"can only extend 2D arrays, not <dim>D")` must be raised with `<dim>` the dimension of the input array.
- If `rows` is smaller the number of rows of `arr`, a `ValueError("invalid rows")` must be raised.
- If `cols` is smaller the number of columns of `arr`, a `ValueError("invalid cols")` must be raised.
- If `fill` is not `None` and not a number, a `ValueError("invalid fill")` must be raised.
- (`rows, cols`) determines the size of the new, extended array, where all elements from `arr` (`aij`) are copied and the new elements (`vi_`, `v_j`, `v__`) are filled as follows.

```
a11 ... a1n v1_ ... v1_
a21 ... a2n v2_ ... v2_
... ... ... ... ... ...
am1 ... amn vm_ ... vm_
v_1 ... v_n v__ ... v__
... ... ... ... ... ...
v_1 ... v_n v__ ... v__
```

- `fill` determines which values used to fill up the extended array.
  - `fill = None`: `vi_`, `v_j`, `v__` are respectively the mean of i^{th} row of `arr`, the mean of j^{th} column of `arr`, the mean of `arr`.
  - `fill != None`: `vi_` = `v_j` = `v__` = `fill`.
- Note that (`rows, cols`) can be equal to the size of `arr`, in which case a new array is still created and `arr` is copied, but there are no new elements that need to be filled.
- The new, extended numpy array must be returned. It contains elements with the same data type of `arr`'s elements.

Hints:

- The function `np.empty_like` or `np.full_like` might be helpful.
- To check if an object is a number, you can use the function `isinstance(object, numbers.Number)`.

Example execution of the programme: Output[a]:

```
m1 = np.arange(2*3).reshape(2,-1)
print(m1)
```
```
[[0 1 2]
 [3 4 5]]
```

```
print(extend(m1, 2, 3))
```
```
[[0 1 2]
 [3 4 5]]
```

```
print(extend(m1, 4, 5))
```
```
[[0 1 2 1 1]
 [3 4 5 4 4]
 [1 2 3 2 2]
 [1 2 3 2 2]]
```

```
try:
    print(extend(m1, 2, 1))
except ValueError as e:
    print(f"ValueError: {e}")
```
```
ValueError: invalid cols
```

```
try:
    print(extend(m1, 1, 2))
except ValueError as e:
    print(f"ValueError: {e}")
```
```
ValueError: invalid rows
```

```
m2 = np.arange(2*3,dtype=float).reshape(2,-1)
print(m2)
```
```
[[0. 1. 2.]
 [3. 4. 5.]]
```

```
print(extend(m2, 4, 5))
```
```
[[0.  1.  2.  1.  1. ]
 [3.  4.  5.  4.  4. ]
 [1.5 2.5 3.5 2.5 2.5]
 [1.5 2.5 3.5 2.5 2.5]]
```

```
print(extend(m1, 4, 5, fill=10))
```
```
[[ 0  1  2 10 10]
 [ 3  4  5 10 10]
 [10 10 10 10 10]
 [10 10 10 10 10]]
```

```
try:
    print(extend(m2, 4,4, fill="foo"))
except ValueError as e:
    print(f"ValueError: {e}")
```
```
ValueError: invalid fill
```

```
m3 = np.ones(1)
print(m3)
```
```
[1.]
```

```
try:
    print(extend(m3, 2, 3))
except ValueError as e:
    print(f"ValueError: {e}")
```
```
ValueError: can only extend 2D arrays, not 1D
```

[a] Empty lines are shown here just for clarity.

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` having any number of dimensions. It transforms the input array `arr` in place by updating the results of `f` for each element directly into `arr`. 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 `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
```

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

Write a function `moving_average_2D(arr: np.ndarray, size: int) -> np.ndarray` that calculates moving average on the input 2D numpy array `arr`. The function works as follows:

- If `arr` is not 2D, a `ValueError(f"apply for 2D array, not <dim>D")` must be raised with `<dim>` the dimension of the input array.
- If elements in `arr` are not numbers, a `TypeError("Invalid data type")` must be raised.
- `size` specifies the size of a window used to calculate the of the elements in `arr`. If `size` is smaller than 1 or greater than the size of any dimension of the `arr`, a `ValueError("Invalid window size")` must be raised.
- The function returns a 2D array of data type `float` of shape `(nr-size+1, nc-size+1)` where `(nr, nc)` is the shape of the input array `arr`.
- The calculations are as follows (a visualization for moving the window can be found here):
  - A window of size `size` x `size` starts at the upper-left corner of `arr` (position `(0,0)`).
  - The window moves down or to the right, stopping when reaching the last row or column where the whole window can fit inside `arr`.
  - At each position `(i,j)` in `arr` where the window is located, the average value of the window elements is calculated.
  - This average value is assigned to the corresponding position `(i,j)` in the output array.

Hints:

- You can use `np.issubdtype(arr.dtype, np.number)` to check if elements of `arr` are numbers.

Example execution of the programme:

```
arr = np.arange(4*5).reshape(4, -1)
print(arr)



result = moving_average_2D(arr, 3)
print(result)

try:
    moving_average_2D(arr, 5)
except ValueError as e:
    print(f"ValueError: {e}")

try:
    moving_average_2D(np.array([["a", "b"], ["c", "d"]]), 2)
except TypeError as e:
    print(f"TypeError: {e}")
```

Output[a]:

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]

[[ 6.  7.  8.]
 [11. 12. 13.]]

ValueError: Invalid window size

TypeError: Invalid data type
```

[a] Empty lines are shown here just for clarity.