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.

Exercise 1 – Submission: `ex1.py` 30 Points

Write a function `extend(arr: np.ndarray, size: int, fill=None) -> np.ndarray` that extends a 1D array `arr` to a given size `size` 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 1D, a `ValueError` must be raised.
- If `size` is smaller than the 1D array, a `ValueError` must be raised.
- `size` determines the size of the new, extended array, where all elements from `arr` are copied and the new elements at the end are filled according to `fill` (see below). More specifically, the new array content is `[a1, a2, ..., an, f, f, ..., f]`, where `ai` are the elements of `arr` and `f` is the fill value. Note that `size` 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 `f` that need to be filled.
- `fill` determines which value is used to fill up the extended array. An arbitrary value can be chosen (you can assume the user chooses a value compatible with the array data type). However, there are two special cases:
  - `fill=None`: In this case, no fill value should be used, i.e., the new, extended array elements might have arbitrary/uninitialized data.
  - `fill="mean"`: If the array has a numeric data type, the mean value of `arr` should be used as fill value. Otherwise (array is not numeric), it is just a regular string fill value.
- The new, extended array must be returned. It must have the same data type as `arr`.

Example program execution:

```
print(extend(np.arange(4), 7))
print(extend(np.arange(4), 7, fill=0))
print(extend(np.arange(4), 7, fill="mean"))
print(extend(np.arange(4, dtype=float), 7, fill="mean"))
print(extend(np.array(["hello", "world"]), 5, "mean"))
```

Example output (first output might vary):

```
[        0         1         2         3 125632899  41684340  42140028]
[0 1 2 3 0 0 0]
[0 1 2 3 1 1 1]
[0.  1.  2.  3.  1.5 1.5 1.5]
['hello' 'world' 'mean' 'mean' 'mean']
```

Hints:

- The functions `np.empty_like` and `np.full_like` might be helpful.
- To determine whether an array `arr` has a numeric data type, you can use the function `np.issubdtype(arr.dtype, np.number)`, according to the NumPy type hierarchy.

Exercise 2 – Submission: ex2.py 20 Points

Write a function `matrix_stats(matrix: np.ndarray) -> dict` that computes summary statistics for a 2D array/matrix. The function works as follows:

- If `matrix` is not 2D, a `ValueError` must be raised.
- The function returns a dictionary with the following entries:
  - `"total_sum`: The sum of all elements of `matrix`.
  - `"row_sums`: A NumPy array containing the sums of each row in `matrix`.
  - `"column_sums`: A NumPy array containing the sums of each column in `matrix`.

Example program execution:

```
print(matrix_stats(np.arange(3 * 4).reshape(3, 4)))
```

Example output:

```
{'total_sum': 66, 'row_sums': array([ 6, 22, 38]),
 'column_sums': array([12, 15, 18, 21])}
```

Exercise 3 – Submission: ex3.py 20 Points

Write a function `create_data(setups: list[dict], seed=None) -> dict` that creates randomized data arrays in the following way:

- `setups` specifies a list of dictionaries, where each such dictionary has the following entries:
  - `"id"`: The ID of the randomized data array to create.
  - `"n"`: The shape of the randomized data array to create. Can be an integer or a tuple of the form `(d1, d2, ..., dn)`, where `di` is a dimension entry.
  - `"a"`: Specifies the lower (inclusive) bound of the range $[a, b)$, from which random elements are sampled.
  - `"b"`: Specifies the upper (exclusive) bound of the range $[a, b)$, from which random elements are sampled.
- For each dictionary in `setup`, a new randomized NumPy array must be created. The array has the shape `n`, and its elements must be drawn from a uniform distribution specified by $[a, b)$. The data type must be some floating point data type (which one does not matter, e.g., both `np.float32` and `np.float64` are fine).
- `seed` is used to initialize the seed of the random number generator.
- The function must return a dictionary where the keys are the respective IDs (`id`) and the values the created arrays (according to `n`, `a` and `b`).

Example program execution:

```
for id_, arr in create_data([
    {"id": "classA", "n": 10, "a": 0, "b": 1.5},
    {"id": "classB", "n": 20, "a": 3, "b": 4},
    {"id": "classC", "n": (5, 10), "a": 0, "b": 10}
], 0).items():
    print(id_, arr.shape)
    print(arr)
```

Example output (randomization might vary depending on the version of NumPy):

```
classA (10,)
[0.95544253 0.40468007 0.06146029 0.02479145 1.21990536 1.36913337
 0.90995366 1.09424484 0.81543749 1.40260864]
classB (20,)
[3.81585355 3.0027385  3.85740428 3.03358558 3.72965545 3.17565562
 3.86317892 3.54146122 3.29971189 3.42268722 3.02831967 3.12428328
 3.67062441 3.64718951 3.61538511 3.38367755 3.99720994 3.98083534
 3.68554198 3.65045928]
classC (5, 10)
[[6.88446731 3.88921424 1.35096505 7.2148834  5.25354322 3.10241876
  4.85835359 8.89487834 9.34043516 3.57795197]
 [5.71529831 3.21869391 5.9430003  3.37911226 3.91619001 8.90274352
  2.27157594 6.23187145 0.84015344 8.32644148]
 [7.87098307 2.39369443 8.76484231 0.58568035 3.36117061 1.50279467
  4.50339367 7.9632427  2.30642209 0.52021301]
 [4.0455184  1.98513045 0.90753046 5.80332386 2.98696133 6.71994878
  1.99515444 9.42113111 3.65110168 1.0549528 ]
 [6.29108152 9.27154553 4.40377155 9.54590494 4.99895814 4.25228625
  6.20213452 9.95096505 9.48943675 4.60045139]]
```

Exercise 4 – Submission: `ex4.py` 30 Points

Write a function

```
create_minefield(
    rows: int,
    cols: int,
    n_mines: int,
    seed=None
) -> np.ndarray
```

that creates an array which could be used to implement the game Minesweeper. In Minesweeper, the goal is to uncover all cells of a 2D grid (=the minefield) that are not mines. Such a minefield can, for instance, be represented as a 2D integer array, where a value of -1 represents a mine cell and a value ≥ 0 a non-mine cell. The value `x` of a non-mine cell indicates how many mines there are in its neighboring cells, where "neighboring" is defined as the 8 surrounding cells (visualized with `-`):

```
- - -
- x -
- - -
```

We can also define the minefield via the mine cells: A mine cell increments the values of all neighboring cells by 1 (except if a neighboring cell is a mine cell itself). Out-of-grid cells are simply ignored.

Either definition is fine, they result in the same minefield output array. Here is an example of a 7×7 minefield with 3 mines (character `m`; left) and the resulting 2D integer array (right):

```
-  -  -  -  -  -  -      [[ 0  0  0  0  0  0  0]
-  -  -  -  -  -  -       [ 0  0  0  0  0  0  0]
-  -  -  -  -  -  -       [ 0  0  0  1  1  1  0]
-  -  -  -  m  -  -  -->  [ 0  1  1  2 -1  1  0]
-  -  m  -  -  -  -       [ 0  1 -1  3  2  2  0]
-  -  -  -  m  -  -       [ 0  1  1  2 -1  1  0]
-  -  -  -  -  -  -       [ 0  0  0  1  1  1  0]]
```

Your task is to implement the above function so that it returns such 2D integer minefield arrays. The function works as follows:

- `rows` specifies the number of rows of the minefield. If `rows` is smaller than 2, a `ValueError` must be raised.
- `cols` specifies the number of columns of the minefield. If `cols` is smaller than 2, a `ValueError` must be raised.
- `n_mines` specifies the number of mines that should be randomly placed on the minefield (=number of mine cells). If `n_mines` is smaller than 1 or ≥ `rows·cols` , a `ValueError` must be raised. Mines must be placed on different cells, i.e., a single cell can only contain a single mine.
- `seed` is used to initialize the seed of the random number generator.
- The function must return a 2D array of data type `int`.

Example function calls and results (randomization might vary depending on the version of NumPy):

```
create_minefield(7, 7, 3, 0) =
[[ 0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0]
 [ 0  0  0  1  1  1  0]
 [ 0  1  1  2 -1  1  0]
 [ 0  1 -1  3  2  2  0]
 [ 0  1  1  2 -1  1  0]
 [ 0  0  0  1  1  1  0]]
```

```
create_minefield(7, 7, 1, 0) =
[[ 0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0]
 [ 0  0  0  0  0  1  1]
 [ 0  0  0  0  0  1 -1]
 [ 0  0  0  0  0  1  1]]
```

```
create_minefield(7, 7, 20, 0) =
[[-1 -1 -1  2  1  1 -1]
 [ 3 -1  5 -1  2  2  2]
 [ 2  3 -1  3  4 -1  2]
 [-1  3  3  4 -1 -1  2]
 [ 2 -1  3 -1 -1  4  1]
 [ 2  3 -1  6 -1  4  1]
 [-1  2  2 -1 -1 -1  1]]
```

```
create_minefield(7, 7, 5, 2) =
[[ 0  0  0  0  2 -1  2]
 [ 1  1  0  0  2 -1  3]
 [-1  1  0  0  1  2 -1]
 [ 1  1  0  0  0  1  1]
 [ 0  1  1  1  0  0  0]
 [ 0  1 -1  1  0  0  0]
 [ 0  1  1  1  0  0  0]]
```

Hints:

- Take a look at the random number generator NumPy documentation for useful methods.
- The implementation is up to you, but try to utilize NumPy's functionality such as broadcasting.