print

June 9, 2022

1 PE Answers

[ ]:

```
##################
### Question 3 ###
##################

## Task A ###
def row_sum(matrix):
    return list(map(lambda row: sum(row), matrix))

def col_sum(matrix):
    # print(matrix)
    if matrix == [] or matrix[0] == []:
        return []

    return [sum(map(lambda x: x[0], matrix))] + col_sum(list(map(lambda x: x[1:], matrix)))

#
# no error checking needed
#
def get_shape(lst):
    shape = [len(lst)]
    tmp = lst[0]

    while isinstance(tmp, list):
        shape.append(len(tmp))
        tmp = tmp[0]

    return shape

def get_value(lst, idx):
    if len(idx) == 1:
        return lst[idx[0]]
    else:
        return get_value(lst[idx[0]], idx[1:])

def set_value(lst, idx, val):  # same as get_value.....
    if len(idx) == 1:
        lst[idx[0]] = val
        return
    else:
```

```python
        return set_value(lst[idx[0]], idx[1:], val)

def create_arr(shape):
    if len(shape) == 1:
        return [0] * shape[0]
    else:
        result = [[]] * shape[0]
        for i in range(shape[0]):
            # print(result[i])
            result[i] = create_arr(shape[1:])

        return result

# get_shape(create_lst([3,4]) gives [3,4] with all values are 0

def next_idx(idx, shape):
    if len(shape) == 0:
        return None  # reach the max already
    else:
        if idx[-1] == shape[-1] - 1:
            temp = next_idx(idx[:-1], shape[:-1])
            if temp is None:  # need this to avoid error...boundary condition can check.
                return None
            else:
                return temp + [0]
        else:
            idx[-1] += 1
            return idx

def test(shape):
    idx = [0] * len(shape)
    while idx is not None:
        idx = next_idx(idx, shape)
        print(idx)

    return

# sum along - tts version
def sum_along(axis, lst):  # axis starts with zero...need to check row and column
    # setting up
    shape = get_shape(lst)
    s = shape.pop(axis)  # remind them

    if len(shape) == 0:  # this is a 1D problem
        return sum(lst)

    result = create_arr(shape)
    rIdx = [0] * len(shape)

    while rIdx is not None:
        val = 0
        for i in range(s):
            idx = rIdx.copy()
            idx.insert(axis, i)
            val += get_value(lst, idx)

        set_value(result, rIdx, val)
        rIdx = next_idx(rIdx, shape)

    return result
```

[ ]:

```
##################
### Question 4 ###
##################
class Matrix(object):

    ## Task A ###
    def __init__(self, nrows, ncols):
        self.dict = {}
        self.nrows = nrows
        self.ncols = ncols

    def get(self, idx):
        return self.dict.get(idx, 0)

    def insert(self, idx, val):
        self.dict[idx] = val

    def delete(self, idx):
        self.dict.pop(idx)

    def dict2list(self):
        res = [[0] * self.ncols for _ in range(self.nrows)]
        for idx in self.dict:
            res[idx[0]][idx[1]] = self.dict[idx]
        return res

    ## Task B ###
    def transpose(self):
        output = Matrix(self.ncols, self.nrows)
        for (idx, value) in self.dict.items():
            output.insert((idx[1], idx[0]), value)

        return output

    ## Task C ###

    def multiply(self, m2):
        output = Matrix(self.nrows, m2.ncols)
        for (i, k) in self.dict:
            for j in range(m2.ncols):
                if (k, j) in m2.dict:
                    output.insert((i, j), output.get((i, j)) + self.get((i, k)) * m2.
↪get((k, j)))
```

```
        return output

    ## For debug ###
    def __str__(self):
        return f'{self.nrows} rows, {self.ncols} cols, {self.dict}'
```

2 Arrays

[4]:

```
import functools
from typing import *
from copy import deepcopy
from operator import mul, add, sub


class Array:
    """A crappy copy of numpy's implementation of ndarray"""

    def __init__(self, shape: Optional[Union[List[int], Tuple[int]]] = None,
                 init: Optional[Union[List[List], List[int]]] = None):

        if all(map(lambda x: x is None, (shape, init))):
            raise ValueError('Either shape or initialisation matrix must be defined')
        self.mat = self._zero_matrix(shape) if init is None else init
        flat = self.flatten()
        if not all(type(x) == int or float for x in flat):
            self.mat = None
            raise ValueError('Array must contain only ints and floats')

    def __call__(self):
        return self.mat

    def __repr__(self):
        return str(self.mat)

    def __str__(self):
        return str(self.mat)

    def __add__(self, other):
        if self.assert_dims(other):
            self_flat = self.flatten(inplace=False)
            other_flat = other.flatten(inplace=False)
            combined = [functools.reduce(add, x) for x in (zip(self_flat, other_flat))]
            new_arr = Array(init=combined)
            new_arr.reshape(dims=self.shape(), inplace=True)
            return new_arr
        else:
            raise ValueError('Both Arrays are not of the same dimension spec')

    def __sub__(self, other):
        if self.assert_dims(other):
            self_flat = self.flatten(inplace=False)
            other_flat = other.flatten(inplace=False)
            combined = [functools.reduce(sub, x) for x in (zip(self_flat, other_flat))]
            new_arr = Array(init=combined)
            new_arr.reshape(dims=self.shape(), inplace=True)
            return new_arr
        else:
            raise ValueError('Both Arrays are not of the same dimension spec')

    def __mul__(self, other):
        """M x A  (*)  A x N == M x N"""

        def go_factor(ls: List, operator: Callable):
            if type(ls) == list:
                if len(ls) > 0:
                    if any(map(lambda x: type(x) == int, ls)):
                        temp = [operator(v) for v in ls]
                        if not all(type(x) == int or float for x in ls):
                            raise ValueError('Cannot input non-integers or non-floats into Array')
                        else:
                            ls[:] = temp
                    else:
                        go_factor(ls[0], operator)
                        go_factor(ls[1:], operator)
            else:
                return

        if isinstance(other, int):
            go_factor(self.mat, operator=lambda x: other * x)
            return Array(init=self.mat)
        elif isinstance(other, Callable) and type(other) != type(self):
            go_factor(self.mat, operator=other)
            return Array(init=self.mat)
        elif isinstance(other, Array):
            self_shape = self.shape()
            other_shape = other.shape()
            if self_shape[-1] != other_shape[0]:
                raise ValueError('Both Arrays of not of compatible dimension spec')
            elif len(self_shape) == len(other_shape) == 2:
                # TODO ONLY WORKS FOR 2D ARRS FOR NOW IDK LOL
                self_mat = self()
                other_mat = other()
                zeros = self._zero_matrix(dims=[self_shape[0], other_shape[-1]])

                # taken from https://www.programiz.com/python-programming/examples/multiply-matrix cuz my math fail
                for i in range(len(self_mat)):
                    for j in range(len(other_mat[0])):
                        for k in range(len(other_mat)):
                            zeros[i][j] += self_mat[i][k] * other_mat[k][j]

                reshaped = Array(init=zeros)
                return reshaped
            else:
                raise ValueError('Array multiplication can only be done on 2D arrays')
        else:
            raise TypeError(f'Cannot multiply array by type<{type(other)}>')

    def __neg__(self):
        """Using inner func for negation"""

        return self.__mul__(-1)

    def _zero_matrix(self, dims: Union[List[int], Tuple[int]]):
        """Returns an empty matrix"""

        if len(dims) == 1:
            arr_shape = dims[0]
            return type(dims)([0 for _ in range(arr_shape)])
        else:
            return [self._zero_matrix(dims[1:]) for _ in range(dims[0])]

    def _modify(self, idx: Union[List[int], Tuple[int]], value: Optional[Any] = None):
        """Inner function to search and modify the idx"""

        # assert dims are the same
        assert len(shape(self.mat)) != len(idx), 'Invalid Dimensions'

        try:
            if len(shape(self.mat)) == 1:
                self.mat[idx[0]] = value
            else:
                current = self.mat[idx[0]]
                print(current)
                idx = idx[1:]

                for i in idx[:-1]:
                    current = current[i]

                current[idx[-1]] = 0 if value is None else value
        except (IndexError, KeyError):
            raise ValueError(f'{idx} is invalid')

    def assert_dims(self, other):
        return functools.reduce(mul, self.shape()) == functools.reduce(mul, other.
↪shape())

    @classmethod
    def arange(cls, start: Union[int, float] = 0, end: Union[float, int] = 0,
               step: Optional[Union[int, float]] = None,
               shape: Optional[Union[List[int], Tuple[int]]] = None):
        """Similar to numpy's arange, generating a range of numbers according to
        step and then reshaping if necessary"""

        if step is None:
            step = type(end)(1)
        all_elements = []

        while start < end:
            all_elements.append(start)
            start += step

        new_arange = Array(init=all_elements)

        if shape is not None:
            new_arange.reshape(dims=shape, inplace=True)

        return new_arange

    def shape(self):
        """Returns the shape of the array"""

        def inner(ls: List[List]):
            if type(ls) != list and type(ls) != tuple:
                return
            else:
                mat_shape.append(len(ls))
                inner(ls[0])

        mat_shape = []
        inner(self.mat)

        if mat_shape is []:
            return 'Invalid Matrix'
        else:
            return mat_shape

    def insert(self, idx: Union[List[int], Tuple[int]], value: Optional[Any] = None):
        """Inserts the item into the array"""
        self._modify(idx, value)

    def delete(self, idx: Union[List[int], Tuple[int]]):
        """Purge an item from the array and replace it by 0"""
        self._modify(idx)

    def reshape(self, dims: Union[List[int], Tuple[int]], inplace: bool = False):
        """Important Array reshaping function"""

        if functools.reduce(mul, dims) != functools.reduce(mul, self.shape()):
            raise ValueError(f'Invalid dimension spec: {dims}')
        else:
            def go_inner(ls: List):
                if type(ls) == list:
                    if len(ls) > 0:
                        if any(map(lambda x: type(x) == int, ls)):
                            ls[:] = [flattened.pop(0) for _ in range(len(ls))]
                        else:
                            go_inner(ls[0])
                            go_inner(ls[1:])
                else:
                    return

            flattened = self.flatten(inplace=False)
            new = self._zero_matrix(dims)
            go_inner(new)

            if inplace:
                self.mat = new
            else:
                return new

    def expand_dims(self, inplace: bool = False):
        """Increases the dimensionality of the array by 1, adding one onto the␣
↪outermost dimension"""
        return self.reshape(dims=[1] + self.shape(), inplace=inplace)

    def sum(self, axis: int):
        """Sums up all elements along a specified axis, lists are concatenated and␣
↪values are added up"""

        if axis < 0 or axis > len(self.shape()) - 1:
            raise ValueError(f'Cannot squeeze to axis {axis}')
        elif len(self.shape()) == 1:
            return [sum(self.mat)]
        else:
            def go_layer(ls: List):
                nonlocal descended_layer

                if type(ls) == list:
                    if len(ls) > 0:
                        if descended_layer == axis:
                            if type(functools.reduce(add, ls)) == int:
                                ls[:] = [functools.reduce(add, ls)]
                                return descended_layer
                            else:
                                ls[:] = functools.reduce(add, ls)
                                # TODO
                                # ls[:] = functools.reduce(add, ls)
                                # print('THING', ls)

                                return descended_layer
                        else:
                            descended_layer += 1
                            traversed = go_layer(ls[0])
                            if traversed is not None:
                                descended_layer -= traversed - 1
                            go_layer(ls[1:])
                else:
                    return

            descended_layer = 0
            go_layer(self.mat)

    def squeeze(self, axis: Optional[int] = None, inplace: bool = False):
        """Destroys all redundant dimensions in the array (all 1s)"""

        shape = self.shape()

        if axis is None:
            shape = list(filter(lambda x: x > 1, shape))
        else:
            shape = shape[:axis] + shape[axis + 1:]
            if functools.reduce(mul, shape) != functools.reduce(mul, self.shape()):
                raise ValueError('Invalid Squeeze Axis: Squeezing along this axis '
                                 'creates an array with the wrong flattened shape')

        if inplace:
            self.reshape(dims=shape, inplace=True)
        else:
            return self.reshape(dims=shape)

    def flatten(self, inplace: bool = False):
        """Flattens the array into 1D"""

        def collapse(arr):
            """Destroys the outermost dimension"""

            return functools.reduce(add, arr)

        flattened_shape = functools.reduce(mul, self.shape())
        copied = deepcopy(self.mat)

        while len(copied) != flattened_shape or type(copied[0]) == list:
            copied = collapse(copied)

        if inplace:
            self.mat = copied
        else:
            return copied

    def transpose(self):
        """Swap the first 2 dimensions"""

        # only works on 2 dims ;-;
        if len(self.shape()) == 2:
            self.mat = list(map(list, zip(*self.mat)))
        elif len(self.shape()) > 2:
            print('Warning: Tranposing >2D Matrix, result is the permutation of the␣
 ↪first 2 axes')
            self.mat = list(map(list, zip(*self.mat)))

    def diagonal(self):
        """
        Returns all diagonals of the array
        Idea taken from:
        https://stackoverflow.com/questions/6313308/
↪get-all-the-diagonals-in-a-matrix-list-of-lists-in-python
        """

        curr_shape = self.shape()
        mat = self.mat
        while curr_shape[0] == 1:
            mat = self.reshape(dims=curr_shape[1:])
            curr_shape = curr_shape[1:]

        max_col = len(mat[0])
        max_row = len(mat)
        cols = [[] for _ in range(max_col)]
        rows = [[] for _ in range(max_row)]
        forward_diag = [[] for _ in range(max_row + max_col - 1)]
        backward_diag = [[] for _ in range(len(forward_diag))]
        min_backwards = -max_row + 1

        for col in range(max_col):
            for row in range(max_row):
                cols[col].append(mat[row][col])
                rows[row].append(mat[row][col])
                forward_diag[row + col].append(mat[row][col])
                backward_diag[col - row - min_backwards].append(mat[row][col])

        diags = {
            'rows': rows,
            'cols': cols,
            'fdiags': forward_diag,
            'bdiags': backward_diag
        }

        return diags

    def rotate(self, degrees: int = 90):
        if degrees % 90 != 0:
            raise ValueError('Degrees must be a multiple of 90')
        else:
            rots = (degrees % 90) + 1
            copied = deepcopy(self.mat)

            for i in range(rots):
                copied = [list(x)[::-1] for x in zip(*copied)]

            self.mat = copied

    def spiral(self):
        new_mat = []
        copied = deepcopy(self.mat)

        while copied:
            new_mat.extend(copied.pop(0))

            for i in range(0, len(copied) - 1):
                if copied[i]:
                    new_mat.append(copied[i].pop())

            if copied:
                new_mat.extend(copied.pop()[::-1])

            for i in range(len(copied) - 1, -1, -1):
                if copied[i]:
                    new_mat.append(copied[i].pop(0))

        return new_mat

    def clear(self, inplace: bool = False):
        """Cleans out the matrix and fill with 0s"""

        if inplace:
            self.mat = self.__mul__(other=0)
        else:
            return self.__mul__(other=0)

    def display(self):
        """Prints out the representation, row by row, with respect to the first␣
↪outermost dimension"""

        for row in self.mat:
            print(row)
```

[8]:

```
# Tests
print('### Shape ###')
mat3 = Array(init=[[[1, 2, 3], [4, 5, 6], [7, 8, 9]]])
print('Shape:', mat3.shape())

print('\n### Reshape ###')
mat3.reshape(dims=[1, 3, 1, 3], inplace=True)
print(mat3)

print('\n### Squeeze ###')
mat3.squeeze(axis=0, inplace=True)
print(mat3, '\tShape:', mat3.shape())

print('\n### Expand Dims ###')
mat3.expand_dims(inplace=True)
print(mat3, '\tShape:', mat3.shape())

print('\n### Negate ###')
mat3 = -mat3
print(mat3)
mat3 = -mat3
print(mat3)
print('\n### Flatten ###')
mat3.flatten(inplace=True)
print(mat3)
mat3.reshape(dims=[1, 3, 3], inplace=True)
print(mat3)

print('\n### Sum Along ###')
mat3.sum(axis=1)
print(mat3)
mat3.reshape(dims=[3, 3, 1], inplace=True)
print(mat3)

print('\n### Transpose ###')
mat3.transpose()
print(mat3)
mat3.transpose()
print(mat3)

print('\n### More Flatten ###')
mat4 = Array(init=[[1], [2], [3], [4]])
print(mat4.shape())
mat4.reshape(dims=[2, 2], inplace=True)
print(mat4.flatten(inplace=True))
print(mat4)

print('\n### Addition ###')
mat0 = Array(init=[[1, 2, 3, 4], [5, 1, 2, 3], [9, 5, 1, 2]])
mat1 = Array(init=[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
mat2 = mat0 + mat1
print(mat2)

print('\n### Subtraction ###')
matminus = mat0 - mat1
print(matminus)

print('\n### Multiplication ###')
test1 = Array(init=[[2], [1], [1]])
test2 = Array(init=[[10, 20, 30]])
print(test1.shape())
print(test2.shape())
print(test1 * test2, '\t\tShape:', (test1 * test2).shape())

print('\n### Diagonal ###')
for item, value in mat0.diagonal().items():
    print(item, value)

print('\n### Clear ###')
test2.clear(inplace=True)
print(test2)

# ew = Array() is an error lol

print('\n### Arange ###')
```

```python
newnew = Array.arange(end=10, shape=[2, 5])
print(newnew)

print('\n### Multiplication with Callable is permitted ###')
print(newnew * (lambda x: x ** 2))

print('\n### Prints the contents of the Array ###')
mat3.display()

print('\n### Rotate ###')
mat3.rotate(degrees=90)
mat3.display()

print('\n### Spiral ###')
print(mat3.spiral())
```

```
### Shape ###
Shape: [1, 3, 3]

### Reshape ###
[[[[1, 2, 3]], [[4, 5, 6]], [[7, 8, 9]]]]

### Squeeze ###
[[[1, 2, 3]], [[4, 5, 6]], [[7, 8, 9]]]         Shape: [3, 1, 3]

### Expand Dims ###
[[[[1, 2, 3]], [[4, 5, 6]], [[7, 8, 9]]]]       Shape: [1, 3, 1, 3]

### Negate ###
[[[[-1, -2, -3]], [[-4, -5, -6]], [[-7, -8, -9]]]]
[[[[1, 2, 3]], [[4, 5, 6]], [[7, 8, 9]]]]

### Flatten ###
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[[[1, 2, 3], [4, 5, 6], [7, 8, 9]]]

### Sum Along ###
[[1, 2, 3, 4, 5, 6, 7, 8, 9]]
[[[1], [2], [3]], [[4], [5], [6]], [[7], [8], [9]]]

### Transpose ###
Warning: Tranposing >2D Matrix, result is the permutation of the first 2 axes
[[[1], [4], [7]], [[2], [5], [8]], [[3], [6], [9]]]
Warning: Tranposing >2D Matrix, result is the permutation of the first 2 axes
[[[1], [2], [3]], [[4], [5], [6]], [[7], [8], [9]]]

### More Flatten ###
[4, 1]
None
[1, 2, 3, 4]

### Addition ###
[[2, 4, 6, 8], [10, 7, 9, 11], [18, 15, 12, 14]]
```

```
### Subtraction ###
[[0, 0, 0, 0], [0, -5, -5, -5], [0, -5, -10, -10]]

### Multiplication ###
[3, 1]
[1, 3]
[[20, 40, 60], [10, 20, 30], [10, 20, 30]]               Shape: [3, 3]

### Diagonal ###
rows [[1, 2, 3, 4], [5, 1, 2, 3], [9, 5, 1, 2]]
cols [[1, 5, 9], [2, 1, 5], [3, 2, 1], [4, 3, 2]]
fdiags [[1], [5, 2], [9, 1, 3], [5, 2, 4], [1, 3], [2]]
bdiags [[9], [5, 5], [1, 1, 1], [2, 2, 2], [3, 3], [4]]

### Clear ###
[[0, 0, 0]]

### Arange ###
[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]

### Multiplication with Callable is permitted ###
[[0, 1, 4, 9, 16], [25, 36, 49, 64, 81]]

### Prints the contents of the Array ###
[[1], [2], [3]]
[[4], [5], [6]]
[[7], [8], [9]]

### Rotate ###
[[7], [4], [1]]
[[8], [5], [2]]
[[9], [6], [3]]

### Spiral ###
[[7], [4], [1], [2], [3], [6], [9], [8], [5]]
```

[ ]: