Special Arrangements of a one dimensional Arrays

```
import numpy as np

# Create an array that starts from integer 1, ends at 20, incremented
by 3.
c = np.arange(1, 20, 3)

lin_spaced_arr = np.linspace(0, 100, 5)
print(lin_spaced_arr)

lin_spaced_arr_int = np.linspace(0, 100, 5, dtype=int) # dtype=float
print(lin_spaced_arr_int)

ones_arr = np.ones(3)   # Return a new array with 3 elements of 1.
print(ones_arr)

zeros_arr = np.zeros(3)   # Return a new array with 3 elements of 0.
print(zeros_arr)

rand_arr = np.random.rand(3)    # elements between 0 and 1 chosen at
random.
print(rand_arr)

[  0.  25.  50.  75. 100.]
[  0  25  50  75 100]
[1. 1. 1.]
[0. 0. 0.]
[0.42457753 0.57112457 0.60284547]

# 1-D array
one_dim_arr = np.array([1, 2, 3, 4, 5, 6])

# Multi-dimensional array using reshape()
multi_dim_arr = np.reshape(one_dim_arr,(2,3))
print(multi_dim_arr)

print(multi_dim_arr.shape)      # Returns shape of 2 rows and 3
columns

print(multi_dim_arr.size)       # Returns total number of elements

[[1 2 3]
 [4 5 6]]
(2, 3)
6
```

**Basic operation on vectors**

- Addition, subtraction, dot product, and cross product
- Different Norms of a vector

```
import numpy as np

# Define two vectors
v1 = np.array([3, 4])
v2 = np.array([1, 2])

v_add = v1 + v2                      # Vector addition
v_sub = v1 - v2                      # Vector subtraction
dot_product1 = v1@v2                 # Dot product
dot_product2 =np.dot(v1,v2)          # Dot product
cross_product = np.cross(v1, v2)     # Cross product
magnitude_v1 = np.linalg.norm(v1)    # Magnitude of a vector

print(f"Vector Addition: {v_add}")
print(f"Vector Subtraction: {v_sub}")
print(f"Dot Product: {dot_product1}")
print(f"Dot Product: {dot_product2}")
print(f"Cross Product: {cross_product}")
print(f"Magnitude of v1: {magnitude_v1}")

# Define a vector
x = np.array([1, 2, 3])

# Compute the Lp norm (p = 2 for Euclidean norm, p = 1 for Manhattan
norm, etc.)
p = 2
norm = np.linalg.norm(x, ord=p)
print(f"L{p} norm of the vector {x} is: {norm}")
```

```
Vector Addition: [4 6]
Vector Subtraction: [2 2]
Dot Product: 11
Dot Product: 11
Cross Product: 2
Magnitude of v1: 5.0
L2 norm of the vector [1 2 3] is: 3.7416573867739413
```

**Find norm of a vector without using "linalg" module**

```
x = np.array([1, 2, 3])

# Manually compute the 2-norm
magnitude_x = np.sqrt((x[0])**2+(x[1])**2+(x[2])**2)  # don't use "^"
despite "**"
print(f"Magnitude of x: {magnitude_x}")

# Manually compute the Lp norm
def norm_p(x, p):
  norm_p = np.sum(np.abs(x) ** p) ** (1 / p)
  return norm_p
```

```
p = 2
print(f"L{p} norm of the vector {x} is: {norm_p(x,p)}")
```

```
Magnitude of x: 3.7416573867739413
L2 norm of the vector [1 2 3] is: 3.7416573867739413
```

**To compute the angle between two vectors**

```
# Define two vectors
v1 = np.array([0, 1])
v2 = np.array([1, 0])

# Compute the angle
cos_theta = (v1@v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))
angle = np.degrees(np.arccos(cos_theta))
print(f"Angle between v1 and v2: {angle:.2f} degrees")
```

```
Angle between v1 and v2: 90.00 degrees
```

**Normalization of vector**

```
# Define a vector
v = np.array([3, 4])

# Normalize the vector
normalized_v = v / np.linalg.norm(v)

print(f"Original Vector: {v}")
print(f"Normalized Vector: {normalized_v}")
```

```
Original Vector: [3 4]
Normalized Vector: [0.6 0.8]
```

**Basic operations on matrices**

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
print(f'Sum of two matrix is: {A+B}')
print(f'Elementwise product of two matrix is: {A*B}')
print(f'Product of two matrix is: {A @ B }')
```

```
Sum of two matrix is: [[ 6  8]
 [10 12]]
Elementwise product of two matrix is: [[ 5 12]
 [21 32]]
Product of two matrix is: [[19 22]
 [43 50]]
```

**Basic operations on matrices in symbolic form**

```
import numpy as np
import sympy as sy
a, b, c, d, e, f, g, h, i, j, k,l=sy.symbols("a, b, c, d, e, f, g, h,
i, j, k, l",real=True,)
A = sy.Matrix([[a, b, c], [d, e, f]])
B = sy.Matrix([[g, h, i], [j, k, l]])
A + B
A - B
A@B.transpose()
sy.eye(5)
```

```
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0],
[0, 0, 1, 0, 0],
[0, 0, 0, 1, 0],
[0, 0, 0, 0, 1]])
```

**Operation between scalars and matrices**

```
# Add a scalar to a matrix
A = np.array([[1, 2], [3, 4]])
A_plus_scalar = A + 10
print("A + 10:\n", A_plus_scalar)

# Multiply each element of a matrix by a scalar
A_times_scalar = A * 2
print("A * 2:\n", A_times_scalar)
```

```
A + 10:
 [[11 12]
 [13 14]]
A * 2:
 [[2 4]
 [6 8]]
```

Transpose, Inverse, Determinant, Rank, Eigen values & vectors of a matrix (Direct)

```
A = np.array([[1, 2], [3, 4]])
print(f'Transpose of matrix:\n  {A}  is: \n{A.transpose()}')
print(f'Inverse of matrix: \n{A} is:\n {np.linalg.inv(A)}')
print(f'Determinant of matrix \n {A} is {np.linalg.det(A)}')
print(f'Rank of matrix \n {A} is {np.linalg.matrix_rank(A)}')

## Eigen values and vectors
lambdas, V = np.linalg.eig(A)
print(f'Eigen values of matrix \n {A} are: \n {lambdas}')
print(f'Eigen vectors of matrix \n {A} are: \n {V}')
```

```
Transpose of matrix:
  [[1 2]
 [3 4]]  is:
[[1 3]
 [2 4]]
Inverse of matrix:
[[1 2]
 [3 4]] is:
 [[-2.   1. ]
 [ 1.5 -0.5]]
Determinant of matrix
 [[1 2]
 [3 4]] is -2.0000000000000004
Rank of matrix
 [[1 2]
 [3 4]] is 2
Eigen values of matrix
 [[1 2]
 [3 4]] are:
 [-0.37228132  5.37228132]
Eigen vectors of matrix
 [[1 2]
 [3 4]] are:
 [[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

**Stacking of matrices**

```
a1 = np.array([[1,1],
               [2,2]])
a2 = np.array([[3,3],
              [4,4]])
print(a1)
print(a2)

# Stack arrays vertically
vert_stack = np.vstack((a1, a2))
print(vert_stack)

# Stack arrays horizontally
horz_stack = np.hstack((a1, a2))
print(horz_stack)
```

```
[[1 1]
 [2 2]]
[[3 3]
 [4 4]]
[[1 1]
 [2 2]
 [3 3]
 [4 4]]
[[1 1 3 3]
 [2 2 4 4]]

# Identity matrix
I = np.eye(3)
print("3x3 Identity Matrix:\n", I)

# Zero matrix
Z = np.zeros((2, 3))
print("2x3 Zero Matrix:\n", Z)

3x3 Identity Matrix:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
2x3 Zero Matrix:
 [[0. 0. 0.]
 [0. 0. 0.]]

# Solving system of linear equation
import numpy as np

# Define A and b
A = np.array([[3, 1], [1, 2]], dtype=np.dtype(int))
b = np.array([9, 8], dtype=np.dtype(int))

# Solve for x
x = np.linalg.solve(A, b)

print(f"Solution x: {x}")

Solution x: [2. 3.]
```

Transpose of a matrix

```
import numpy as np

def transpose(matrix):
    m ,n = matrix.shape
    tran_matrix = np.zeros((m,n))
    for i in range(n):
      for j in range(m):
          tran_matrix[j,i] = matrix[i,j]

    return tran_matrix

B = np.array([[2, 1, 3],
              [7, 4, 9],
              [4, 1, 5]], dtype=int)
print(f"Transpose of matrix B is : \n {transpose(B)}")
```

```
Transpose of matrix B is :
 [[2. 7. 4.]
 [1. 4. 1.]
 [3. 9. 5.]]
```

**Function to compute the determinant of a square matrix (recursive)**

```
import numpy as np

def determinant(matrix):
    n = len(matrix)
    if n == 1:
        return matrix[0][0]  # 1x1 determinant
    if n == 2:  # Base case for 2x2 matrix
        return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1]
[0]

    det = 0
    for col in range(n):
        # Create submatrix for cofactor expansion
        submatrix = [row[:col] + row[col + 1:] for row in matrix[1:]]
        cofactor = ((-1) ** col) * matrix[0][col] *
determinant(submatrix)
        det += cofactor
    return det

# Function to construct the characteristic polynomial
def characteristic_polynomial(matrix):
    import sympy as sp
    n = len(matrix)
    lambd = sp.symbols('lambda')

    # Construct A - lambda * I
    char_matrix = [[matrix[i][j] - (lambd if i == j else 0) for j in
range(n)] for i in range(n)]
    # Compute determinant of (A - lambda * I)
    det = determinant(char_matrix)
    return sp.expand(det)  # Expand to simplify the polynomial

# Example: 3x3 matrix
A = [[6, 0, 0],
     [2, 3, 0],
     [0, 0, 1]]

# Get characteristic polynomial
char_poly = characteristic_polynomial(A)
print("Characteristic Polynomial:", char_poly)
```

```
# Solve for eigenvalues
import sympy as sp
eigenvalues = sp.solvers.solve(char_poly)
print("Eigenvalues:", eigenvalues)

Characteristic Polynomial: -lambda**3 + 10*lambda**2 - 27*lambda + 18
Eigenvalues: [1, 3, 6]
```

**Set up three functions corresponding to the discussed above elementary operations.**

- Multiply any row by non-zero number
- Add two rows and exchange one of the original rows with the result of the addition
- Swap rows

```
def MultiplyRow(M, row_num, row_num_multiple):
    # .copy() function is required here to keep the original matrix
without any changes
    M_new = M.copy()
    # exchange row_num of the matrix M_new with its multiple by
row_num_multiple
    # Note: for simplicity, you can drop check if  row_num_multiple
has non-zero value, which makes the operation valid
    M_new[row_num] = M_new[row_num] * row_num_multiple
    return M_new

def AddRows(M, row_num_1, row_num_2, row_num_1_multiple):
    M_new = M.copy()
    # multiply row_num_1 by row_num_1_multiple and add it to the
row_num_2,
    # exchanging row_num_2 of the matrix M_new with the result
    M_new[row_num_2] += M_new[row_num_1] * row_num_1_multiple
    return M_new

def SwapRows(M, row_num_1, row_num_2):
    M_new = M.copy()
    # exchange row_num_1 and row_num_2 of the matrix M_new
    M_new[row_num_1] = M_new[row_num_1] + M_new[row_num_2]
    M_new[row_num_2] = M_new[row_num_1] - M_new[row_num_2]
    M_new[row_num_1] = M_new[row_num_1] - M_new[row_num_2]
    return M_new


A_test = np.array([
        [1, -2, 3, -4],
        [-5, 6, -7, 8],
        [-4, 3, -2, 1],
        [8, -7, 6, -5]
    ], dtype=np.dtype(float))
print("Original matrix:")
print(A_test)

print("\nOriginal matrix after its third row is multiplied by -2:")
print(MultiplyRow(A_test,2,-2))

print("\nOriginal matrix after exchange of the third row with the sum
of itself and first row multiplied by 4:")
print(AddRows(A_test,0,2,4))

print("\nOriginal matrix after exchange of its first and third rows:")
print(SwapRows(A_test,0,2))
```

```
Original matrix:
[[ 1. -2.  3. -4.]
 [-5.  6. -7.  8.]
 [-4.  3. -2.  1.]
 [ 8. -7.  6. -5.]]

Original matrix after its third row is multiplied by -2:
[[ 1. -2.  3. -4.]
 [-5.  6. -7.  8.]
 [ 8. -6.  4. -2.]
 [ 8. -7.  6. -5.]]

Original matrix after exchange of the third row with the sum of itself
and first row multiplied by 4:
[[  1.  -2.   3.  -4.]
 [ -5.   6.  -7.   8.]
 [  0.  -5.  10. -15.]
 [  8.  -7.   6.  -5.]]

Original matrix after exchange of its first and third rows:
[[-4.  3. -2.  1.]
 [-5.  6. -7.  8.]
 [ 1. -2.  3. -4.]
 [ 8. -7.  6. -5.]]
```

**Apply elementary operations to the defined above matrix A, performing row reduction according to the given instructions.**

**Note**: Feel free to add a return statement between the different matrix operations in the code to check on your results while you are writing the code (commenting off the rest of the function). This way you can see, whether your matrix operations are performed correctly line by line (don't forget to remove the return statement afterwards!).

- to swap row 1 and row 2 of matrix A, use the code SwapRows(A,1,2)
- to multiply row 1 of matrix A by 4 and add it to the row 2, use the code AddRows (A,1,2,4)
- to multiply row 2 of matrix A by 5, use the code MultiplyRow(A,2,5)

```
def augmented_to_ref(A, b):
    # stack horizontally matrix A and vector b, which needs to be
reshaped as a vector (4, 1)
    A_system = np.hstack((A, b.reshape(4, 1)))

    # swap row 0 and row 1 of matrix A_system (remember that indexing
in NumPy array starts from 0)
    A_ref = SwapRows(A_system, 0, 1)

    # multiply row 0 of the new matrix A_ref by -2 and add it to the
row 1
    A_ref = AddRows(A_ref, 0, 1, -2)

    # add row 0 of the new matrix A_ref to the row 2, replacing row 2
    A_ref[2] += A_ref[0]

    # multiply row 0 of the new matrix A_ref by -1 and add it to the
row 3
    A_ref = AddRows(A_ref, 0, 3, -1)

    # add row 2 of the new matrix A_ref to the row 3, replacing row 3
    A_ref[3] += A_ref[2]

    # swap row 1 and 3 of the new matrix A_ref
    A_ref = SwapRows(A_ref, 1, 3)

    # add row 2 of the new matrix A_ref to the row 3, replacing row 3
    A_ref[3] += A_ref[2]

    # multiply row 1 of the new matrix A_ref by -4 and add it to the
row 2
    A_ref = AddRows(A_ref, 1, 2, -4)

    # add row 1 of the new matrix A_ref to the row 3, replacing row 3
    A_ref[3] += A_ref[1]

    # multiply row 3 of the new matrix A_ref by 2 and add it to the
row 2
    A_ref = AddRows(A_ref, 3, 2, 2)

    # multiply row 2 of the new matrix A_ref by -8 and add it to the
row 3
    A_ref = AddRows(A_ref, 2, 3, -8)

    # multiply row 3 of the new matrix A_ref by -1/17
    A_ref[3] = A_ref[3] * -1/17


    return A_ref
A = np.array([
        [2, -1, 1, 1],
        [1, 2, -1, -1],
        [-1, 2, 2, 2],
        [1, -1, 2, 1]
    ], dtype=np.dtype(float))
b = np.array([6, 3, 14, 8], dtype=np.dtype(float))

A_ref = augmented_to_ref(A, b)

print(A_ref)
```

```
[[ 1.  2. -1. -1.  3.]
 [ 0.  1.  4.  3. 22.]
 [ 0.  0.  1.  3.  7.]
 [-0. -0. -0.  1.  1.]]
```

full proper code for finding inverse of a matrix by gauss-jordan elimination

```
import numpy as np

def manual_matrix_inverse(matrix):
    """
    Find the inverse of a square matrix manually using Gauss-Jordan 
elimination.

    Parameters:
        matrix (ndarray): A square numpy array.

    Returns:
        ndarray: The inverse of the input matrix, if it exists.
    """
    n = matrix.shape[0]

    if matrix.shape[0] != matrix.shape[1]:
        raise ValueError("Input must be a square matrix.")

    # Create an augmented matrix [A|I], where I is the identity matrix
    augmented_matrix = np.hstack((matrix, np.eye(n)))

    # Perform row operations to transform [A|I] into [I|A^-1]
    for i in range(n):
        # Make the diagonal element 1
        diag_element = augmented_matrix[i, i]
        if diag_element == 0:
            # Find a row below to swap
            for j in range(i + 1, n):
                if augmented_matrix[j, i] != 0:
                    augmented_matrix[[i, j]] = augmented_matrix[[j, 
i]]
                    diag_element = augmented_matrix[i, i]
                    break
            else:
                raise ValueError("Matrix is singular and cannot be
inverted.")

        augmented_matrix[i] = augmented_matrix[i] / diag_element

        # Make all other elements in the current column 0
        for j in range(n):
            if i != j:
                row_factor = augmented_matrix[j, i]
                augmented_matrix[j] -= row_factor *
augmented_matrix[i]

    # Extract the right-hand side of the augmented matrix, which is
now the inverse
    inverse_matrix = augmented_matrix[:, n:]
    return inverse_matrix

# Example usage
matrix = np.array([[2, 1],
                   [7, 4]], dtype=float)

try:
    inverse = manual_matrix_inverse(matrix)
    print("Inverse of the matrix:")
    print(inverse)
except ValueError as e:
    print(e)
```

```
Inverse of the matrix:
[[ 4. -1.]
 [-7.  2.]]
```

Singular Value Decomposition (SVD) As on slides, SVD of matrix A is:

$$A = U D V^T$$

Where:

- U is an orthogonal m*m matrix; its columns are the left-singular vectors of A
- V is an orthogonal n*n matrix; its columns are the right-singular vectors of A
- D is a diagonal matrix; elements along its diagonal are the singular values of A .

```
A = np.array([[-1, 2], [3, -2], [5, 7]])
U, d, VT = np.linalg.svd(A) # V is already transposed
print("U:\n", U)
print("d:\n", d)
print("VT:\n", VT)
```

```
U:
 [[ 0.12708324  0.47409506  0.87125411]
 [ 0.00164602 -0.87847553  0.47778451]
 [ 0.99189069 -0.0592843  -0.11241989]]
d:
 [8.66918448 4.10429538]
VT:
 [[ 0.55798885  0.82984845]
 [-0.82984845  0.55798885]]
```

$LU$ decomposition is a matrix factorization technique where a matrix $A$ is expressed as the product of a lower triangular matrix $L$ and an upper triangular matrix $U$, i.e., $A=LU$ where:

- L: Lower triangular matrix with ones on the diagonal.
- U: Upper triangular matrix.

```
import numpy as np
import scipy
# Example matrix
A = np.array([[4, 3],
              [6, 3]])

# Perform LU decomposition
P, L, U = scipy.linalg.lu(A)

print("P (Permutation Matrix):")
print(P)
print("L (Lower Triangular Matrix):")
print(L)
print("U (Upper Triangular Matrix):")
print(U)

P (Permutation Matrix):
[[0. 1.]
 [1. 0.]]
L (Lower Triangular Matrix):
[[1.         0.        ]
 [0.66666667 1.        ]]
U (Upper Triangular Matrix):
[[6. 3.]
 [0. 1.]]
```