Numerical Computing (NumPy) - Problem set

Solution 1: Manipulating a 3D Array

```python
import numpy as np

np.random.seed(42)  # For reproducibility
arr_3d = np.random.randint(1, 101, size=(3, 3, 3))

# Replace values divisible by 3 with -1
arr_3d[arr_3d % 3 == 0] = -1
print("Modified array:\n", arr_3d)

# Calculate the mean value for each 2D layer
mean_layer = arr_3d.mean(axis=(1, 2))
print("Mean of each layer:", mean_layer)
```

Solution 2: Boolean Indexing and Filtering

```python
import numpy as np

def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(np.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

arr = np.random.randint(1, 51, 20)

# Identify prime numbers
prime_mask = np.array([is_prime(x) for x in arr])
print("Prime numbers in array:", arr[prime_mask])

# Replace prime numbers with 0
arr[prime_mask] = 0
print("Array after replacing primes:", arr)
```

Solution 3: Array Manipulation and Reshaping

```python
import numpy as np

# Step 1: Create a 1D array of size 30 with random integers between 10 and 100
arr = np.random.randint(10, 100, size=30)
print("Original Array:", arr)

# Step 2: Reshape the array into a 5x6 matrix
matrix = arr.reshape(5, 6)
print("\nReshaped 5x6 Matrix:\n", matrix)
```

```
# Step 3: Compute the sum of the values in each row
row_sums = matrix.sum(axis=1)
print("\nSum of values in each row:", row_sums)

# Step 4: Compute the sum of the values in each column
col_sums = matrix.sum(axis=0)
print("Sum of values in each column:", col_sums)
```

Solution 4: Matrix Determinant and Inverse

```
import numpy as np

matrix = np.random.randint(1, 11, size=(4, 4))
determinant = np.linalg.det(matrix)
print("Matrix:\n", matrix)
print("Determinant:", determinant)

# If determinant is non-zero, calculate the inverse
if np.abs(determinant) > 1e-6:  # Avoid division by very small numbers
    inverse_matrix = np.linalg.inv(matrix)
    print("Inverse matrix:\n", inverse_matrix)

    # Verify that A * A^-1 is the identity matrix
    identity_check = np.dot(matrix, inverse_matrix)
    print("Product (should be identity matrix):\n", identity_check)
else:
    print("Matrix is singular, cannot compute inverse.")
```

Solution 5: Simulating an Ornstein-Uhlenbeck Process

```
import numpy as np
import matplotlib.pyplot as plt

theta = 0.7
mu = 0.05
sigma = 0.15
dt = 0.01
X0 = 0.05
n_steps = 500

# Simulate Ornstein-Uhlenbeck process
X = np.zeros(n_steps)
X[0] = X0
for t in range(1, n_steps):
    X[t] = X[t-1] + theta * (mu - X[t-1]) * dt + sigma * np.sqrt(dt) *
np.random.randn()

# Plot the results
plt.plot(X)
plt.title("Ornstein-Uhlenbeck Process")
plt.xlabel("Time step")
```

```
plt.ylabel("Value")
plt.show()
```

Solution 6: Element-Wise Operations on Multiple Arrays

```
import numpy as np

A = np.random.randint(1, 11, 1000)
B = np.random.randint(1, 11, 1000)

# Find indices where A[i] > B[i]
indices = np.where(A > B)
print("Indices where A > B:", indices)

# Create array C based on the condition
C = np.where(A > B, A, (A + B) / 2)
print("Array C:", C)
```

Solution 7: Matrix Operations and Transposing

```
import numpy as np

# Step 1: Create a 4x4 matrix filled with random integers between 1 and 20
matrix = np.random.randint(1, 21, size=(4, 4))
print("Original Matrix:\n", matrix)

# Step 2: Transpose the matrix
transposed_matrix = matrix.T
print("\nTransposed Matrix:\n", transposed_matrix)

# Step 3: Compute the sum of elements in the original matrix
original_sum = matrix.sum()
print("\nSum of elements in the original matrix:", original_sum)

# Step 4: Compute the sum of elements in the transposed matrix
transposed_sum = transposed_matrix.sum()
print("Sum of elements in the transposed matrix:", transposed_sum)

# Step 5: Verify if the sums are equal
if original_sum == transposed_sum:
    print("\nThe sums of the original and transposed matrix are equal.")
else:
    print("\nThe sums of the original and transposed matrix are not equal.")
```

Solution 8: Advanced Matrix Operations and Broadcasting

```
import numpy as np

# Step 1: Create two 3x3 matrices filled with random integers between 1 and 10
A = np.random.randint(1, 11, size=(3, 3))
B = np.random.randint(1, 11, size=(3, 3))
```

```
print("Matrix A:\n", A)
print("Matrix B:\n", B)

# Step 2: Add the two matrices together using broadcasting
matrix_sum = A + B
print("\nSum of A and B:\n", matrix_sum)

# Step 3: Multiply the two matrices element-wise
elementwise_product = A * B
print("\nElement-wise multiplication of A and B:\n", elementwise_product)

# Step 4: Compute the matrix product of A and B using matrix multiplication
matrix_product = np.dot(A, B)
print("\nMatrix product of A and B:\n", matrix_product)

# Step 5: Subtract the transpose of matrix B from matrix A
subtracted_matrix = A - B.T
print("\nA minus the transpose of B:\n", subtracted_matrix)
```

Solution 9: Function Application with Vectorization

```
import numpy as np

# Step 1: Create a 1D array of size 20 with random integers between -10 and 10
arr = np.random.randint(-10, 11, size=20)
print("Original Array:\n", arr)

# Step 2: Define the function f(x) = 3x^2 + 2x - 5
def f(x):
    return 3*x**2 + 2*x - 5

# Step 3: Apply the function to each element of the array using vectorization
result = f(arr)
print("\nResult after applying f(x):\n", result)
```

Solution 10: Advanced Broadcasting and Boolean Filtering

```
import numpy as np

# Step 1: Create a 6x6 matrix with random integers between 1 and 100
matrix = np.random.randint(1, 101, size=(6, 6))
print("Original Matrix:\n", matrix)

# Step 2: Find all elements divisible by both 3 and 5
divisible_by_3_and_5 = (matrix % 3 == 0) & (matrix % 5 == 0)
print("\nBoolean mask for elements divisible by both 3 and 5:\n",
divisible_by_3_and_5)

# Step 3: Replace such elements with -1 using boolean filtering
matrix[divisible_by_3_and_5] = -1
print("\nModified Matrix:\n", matrix)
```