Data Toolkit Practice Question

Q1:Demonstrate three different methods for creating identical 2D arrays in NumPy. Provide the code for each method and the final output after each method

Ans:

Here are three different methods for creating identical 2D arrays in NumPy, along with the code and resulting output for each method:

Method 1: Using np.array() with a list of lists

code

```
import numpy as np

# Create a list of lists containing the desired array elements
data = [[1, 2, 3], [4, 5, 6]]

# Convert the list of lists to a NumPy array
array_1 = np.array(data)

# Print the resulting array
print("Array 1 (from list of lists):")
print(array_1)
```

Output:

```
Array 1 (from list of lists):
[[1 2 3]
 [4 5 6]]
```

Method 2: Using np.full()

Python

```python
# Define the number of rows and columns
rows = 2
cols = 3

# Create an array filled with a specific value (7 in this case)
array_2 = np.full((rows, cols), 7)

# Print the resulting array
print("\nArray 2 (filled with 7):")
print(array_2)
```

Output:

```
Array 2 (filled with 7):
[[7 7 7]
 [7 7 7]]
```

Method 3: Using np.zeros()

Python

```python
# Define the number of rows and columns
rows = 2
cols = 3

# Create an array filled with zeros
array_3 = np.zeros((rows, cols))

# Print the resulting array
print("\nArray 3 (filled with zeros):")
print(array_3)
```

Output:

```
Array 3 (filled with zeros):
[[0. 0. 0.]
 [0. 0. 0.]]
```

All three methods create a 2D NumPy array with the dimensions (2 rows, 3 columns).

Q2:Using the Numpy function, generate an array of 100 evenly spaced numbers between 1 and 10 and Reshape that 1D array into a 2D array

Ans:

code:

```
# Generate 100 evenly spaced numbers between 1 and 10
evenly_spaced_numbers = np.linspace(1, 10, 100)

# Reshape the 1D array into a 10x10 2D array
reshaped_array = evenly_spaced_numbers.reshape(10, 10)

# Print the original and reshaped arrays
print("Original 1D array:")
print(evenly_spaced_numbers)

print("\nReshaped 2D array:")
print(reshaped_array)
```

This code will output the following:

```
Original 1D array:
[ 1.         1.09090909  1.18181818  1.27272727  1.36363636  1.45454545
  1.54545455  1.63636364  1.72727273  1.81818182  1.90909091  2.
  2.09090909  2.18181818  2.27272727  2.36363636  2.45454545  2.54545455
  2.63636364  2.72727273  2.81818182  2.90909091  3.         3.09090909
  3.18181818  3.27272727  3.36363636  3.45454545  3.54545455  3.63636364
  3.72727273  3.81818182  3.90909091  4.         4.09090909  4.18181818
  4.27272727  4.36363636  4.45454545  4.54545455  4.63636364  4.72727273
```

```
 4.81818182  4.90909091  5.          5.09090909  5.18181818  5.27272727
 5.36363636  5.45454545  5.54545455  5.63636364  5.72727273  5.8
```

Q3:Explain the following terms
.The difference in np.array, np.asarray and np.asanyarray
.The difference between Deep copy and shallow copy

Ans:
Differences between `np.array`, `np.asarray`, and `np.asanyarray`:

These functions in NumPy all serve the purpose of converting data into NumPy arrays, but they have subtle nuances:

- `np.array`:
    - The most versatile and commonly used function for creating NumPy arrays.
    - Accepts a wide range of data structures as input, including lists, tuples, and other arrays.
    - **Creates a new NumPy array by default.** This means it copies the data into a new memory location, even if the input is already a NumPy array. This can be useful when you want to ensure you're working with a fresh copy of the data or when modifying the array might have unintended consequences on the original data.
- `np.asarray`:
    - Similar to `np.array` but aims to be more memory-efficient.
    - If the input is already a NumPy array, `np.asarray` typically avoids copying the data and returns a view of the original array. This is a reference to the same underlying data, so changes made to the returned array will also affect the original array.

    - If the input is not a NumPy array, `np.asarray` behaves similarly to `np.array` and creates a new copy.
- `np.asanyarray`:
    - Offers the most flexibility for converting data to arrays.
    - Like `np.asarray`, it tries to avoid copying data if the input is already a NumPy array and returns a view.
    - However, `np.asanyarray` also allows conversion of certain non-array-like objects (e.g., scalars) into NumPy arrays, which `np.array` and `np.asarray` generally wouldn't handle. This can be useful for ensuring consistent array-like behavior for various data types.

Choosing the Right Function:

- In most cases, `np.array` is a safe and reliable choice, especially if you're unsure of the input type or need a guaranteed copy.
- If memory efficiency is a concern, and you're confident the input might already be a NumPy array, use `np.asarray` to potentially avoid unnecessary copying.
- If you need to convert a wider range of data types (including scalars) or want to explicitly allow views, consider `np.asanyarray`.

Deep Copy vs. Shallow Copy:

- **Deep Copy:** Creates a completely independent copy of the data structure, including any nested elements. Changes made to the copy do not affect the original data. This is typically achieved by recursively copying all elements and their values.
- **Shallow Copy:** Creates a new reference to the existing data structure. If the original data contains references to other objects, the copy will also hold those

same references. Modifications made to the copy can potentially affect the original data through these shared references.

Here's a table summarizing the key points:

Feature	`np.array`	`np.asarray`	`np.asanyarray`	Deep Copy	Shallow Copy
Input types	Wide range	Wide range	Most data types	Any data structure	Any data structure
Creates copy	Yes (default)	Avoids if possible	Avoids if possible	Yes	No
Reference	New array	View (if possible)	View (if possible)	New data structure	New reference
Use case	Versatile, safe copy	Memory efficiency (potential view)	Flexible conversion	Independent copy	Efficient for references

Example:

Python

```python
import numpy as np

data_list = [1, 2, 3]
arr1 = np.array(data_list)  # Creates a new copy of the data (deep copy)
arr2 = np.asarray(data_list)  # May create a view (shallow copy if
data_list is a NumPy array)
arr3 = np.asanyarray(data_list)  # May create a view (shallow copy if
data_list is a NumPy array)

# Modify arr1
arr1[0] = 10

# Check if original data is modified
```

```
print("Original data:", data_list)  # Output: Original data: [1, 2, 3]
(unchanged)

# Check if views were created
print("arr2 is view:", arr2.base is data_list)
```

Q4:Generate a 3x3 array with random floating-point numbers between 5 and 20. Then, round each number in the array to 2 decimal places.

Ans:

```
# Generate random numbers between 5 and 20 (exclusive)
random_array = np.random.rand(3, 3) * (20 - 5) + 5

# Round each number to 2 decimal places
rounded_array = np.around(random_array, decimals=2)

# Print the original and rounded arrays
print("Original array:")
print(random_array)

print("\nRounded array:")
print(rounded_array)
```

Q5:Create a NumPy array with random integers between 1 and 10 of shape (5, 6). After creating the array perform the following operations:

a)Extract all even integers from array.

b)Extract all odd integers from array

Ans:

```
    import numpy as np

# Generate a random integer array
array = np.random.randint(1, 11, size=(5, 6))  # 1 to 10 (inclusive)

# Extract even integers
even_integers = array[array % 2 == 0]
```

```
# Extract odd integers
odd_integers = array[array % 2 != 0]

# Print the original array, even integers, and odd integers
print("Original array:")
print(array)

print("\nEven integers:")
print(even_integers)

print("\nOdd integers:")
print(odd_integers)
```

Q7: Clean and transform the 'Phone' column in the sample dataset to remove non-numeric characters and convert it to a numeric data type. Also display the table attributes and data types of each column

Ans:

```
# Assuming you have your DataFrame loaded as 'df'

# Clean and transform the 'Phone' column
df['Phone'] = df['Phone'].str.replace(r'\D', '', regex=True).astype(int)  # Remove
non-numeric with regex, convert to int

# Display table attributes (assuming 'df' is your DataFrame)
print(df.info())

# Display data types of each column
print(df.dtypes)
```

Q9: Filter and select rows from the People_Dataset, where the “Last Name' column contains the name 'Duke', 'Gender' column contains the word Female and ‘Salary’ should be less than 85000

Ans:

```
    import pandas as pd
```

```
# Assuming you have your People_Dataset in a DataFrame 'df'

# Filter rows where Last Name is 'Duke', Gender is 'Female', and Salary is less than
85000
filtered_df = df[(df['Last Name'] == 'Duke') & (df['Gender'] == 'Female') & (df['Salary'] <
85000)]

# Display the filtered DataFrame
print(filtered_df)
```

Q12: Perform the following operations using people data set:
a) Delete the 'Email', 'Phone', and 'Date of birth' columns from the dataset.
b) Delete the rows containing any missing values.
c) Print the final output also

Ans:

```
        import pandas as pd
import numpy as np  # Optional, for handling potential missing value errors

# Assuming you have your people data loaded into a DataFrame named 'df'

# a) Delete specific columns
columns_to_delete = ['Email', 'Phone', 'Date of birth']
df.drop(columns_to_delete, inplace=True)  # Modifies the original DataFrame

# c) Delete rows with missing values (handle potential errors)
try:
  df.dropna(inplace=True)  # Attempt to drop rows with missing values
except KeyError:  # Handle potential error if 'dropna' raises a KeyError
  print("Warning: 'dropna' might not be applicable if there are no missing values in the
DataFrame.")

# d) Print the final output
print(df)
```