upGrad

Lecture Notes:
Python Data Science Libraries

NumPy Arrays

Key Definitions & Concepts

`ndarray`: A multi-dimensional array object with homogeneous data type, the core of NumPy.

Axis: Indicates the dimension along which an operation is applied. For 2D arrays:

- `axis=0`: down columns
- `axis=1`: across rows

Vectorised Operations: Operations applied directly to entire arrays without the need for explicit loops. These are optimised for performance and are a key feature of NumPy.

Universal Functions (ufuncs): Element-wise operations that are vectorised across arrays — e.g., `np.exp()`, `np.sqrt()`

Broadcasting: Allows arrays of different shapes to be combined in arithmetic operations without explicit looping.

Copy vs View: Slices often return views (not copies), meaning modifying the slice may alter the original array.

Core Ideas & Theoretical Intuition

- NumPy leverages **vectorisation** and **memory-efficient storage** to outperform Python lists in numerical tasks.
- Think of **broadcasting** as an implicit replication mechanism. NumPy aligns shapes without needing loops or `.repeat()`.
- Reshaping and transposition are **non-destructive operations** that offer powerful ways to prepare arrays for computation.

- Images and tabular data can be treated as **2D or 3D tensors**, making NumPy the go-to tool for low-level data prep.

Mathematical Foundation

Let:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 10 & 20 \end{bmatrix}$$

- **Broadcasted Addition**:

$$A + B = \begin{bmatrix} 1+10 & 2+20 \\ 3+10 & 4+20 \end{bmatrix} = \begin{bmatrix} 11 & 22 \\ 13 & 24 \end{bmatrix}$$

- **Dot Product**:

$$\text{If } x = [1, 2], \quad y = [3, 4], \quad \text{then } x \cdot y = 1 * 3 + 2 * 4 = 11$$

Implementation

Array Creation

```python
np.array([1, 2, 3])                       # From list

np.zeros((2, 3))                          # 2x3 zeros

np.ones((3,))                             # 1D ones

np.arange(0, 10, 2)                       # [0, 2, 4, 6, 8]

np.linspace(0, 1, 5)                      # [0. , 0.25, ..., 1.]

np.eye(3)                                 # 3x3 identity matrix
```

Indexing and Slicing

```python
a = np.array([[10, 20, 30], [40, 50, 60]])

a[1, 2]         # 60

a[:, 1]         # [20, 50]

a[0:2, 0:2]     # subarray

a[-1, :]        # last row
```

Reshaping & Flattening

```python
a = np.arange(12)

a.reshape(3, 4)         # new shape

a.flatten()             # copy as 1D

a.ravel()               # view as 1D
```

Aggregations

```python
a = np.array([[1, 2], [3, 4]])

a.sum(), a.mean(), a.std()

a.sum(axis=0)             # column-wise

a.max(axis=1)             # row-wise
```

Array Manipulations

`np.append()`: Add elements to an array

Adds values to the end along the specified axis or flattens if `axis=None`

```python
a = np.array([[1, 2], [3, 4]])

np.append(a, [[5, 6]], axis=0)   # Append row → shape (3, 2)

np.append(a, [[7], [8]], axis=1) # Append column → shape (2, 3)
```

Returns a **new array**, and the original is not modified.

`np.delete()`: Remove elements along an axis

Removes rows/columns by index

```python
a = np.array([[1, 2, 3], [4, 5, 6]])

# Remove 2nd row → shape (1, 3)

np.delete(a, 1, axis=0)

# Remove 1st and 3rd columns → shape (2, 1)

np.delete(a, [0, 2], axis=1)
```

Like `append`, it returns a **copy**, not in-place modification.

`.T` or `np.transpose()`: Transpose array dimensions

```python
a = np.array([[1, 2], [3, 4]])

a.T                        # [[1, 3], [2, 4]]

np.transpose(a)            # Same as a.T for 2D
```

For 3D+, use the `axes` argument:

```python
a = np.ones((2, 3, 4))

np.transpose(a, (1, 0, 2))  # Rearranges axes
```

`np.concatenate()`: Join arrays along an axis

Concatenates arrays with compatible dimensions

```python
a = np.array([[1, 2], [3, 4]])

b = np.array([[5, 6]])



np.concatenate((a, b), axis=0)  # Vertical → shape (3, 2)

np.concatenate((a, a), axis=1)  # Horizontal → shape (2, 4)
```

Dimensions must **match on all axes except the one concatenated**.

Axis Swapping

```python
a = np.array([[1, 2], [3, 4]])

a.T                      # transpose: [[1, 3], [2, 4]]

a.swapaxes(0, 1)         # swap dimensions
```

Stacking & Splitting

```python
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

np.stack((a, b))          # shape: (2, 3)
np.hstack((a, b))         # shape: (6,)
np.vstack((a, b))         # shape: (2, 3)

c = np.arange(9).reshape(3, 3)
np.hsplit(c, 3)           # three column arrays
```

Element-wise Operations (Broadcasting)

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

a + 10                     # [11, 12, 13]

a * 2                      # [2, 4, 6]

a ** 2                     # [1, 4, 9]

np.exp(a)                  # [2.71, 7.39, 20.09]
```

Logical Operations & Boolean Indexing

```python
a = np.array([1, 2, 3, 4, 5])
a[a > 3]                        # [4, 5]
np.where(a % 2 == 0, 0, 1) # Replace even with 0
```

Image Manipulation

```python
from PIL import Image
img = Image.open("image.jpg").convert("L")
arr = np.array(img)

# Invert grayscale
inverted = 255 - arr

# Brighten image
bright = np.clip(arr + 30, 0, 255)
```

Real-world Use Cases

- **Data Preprocessing**: Clean and transform numeric datasets before modelling
- **Scientific Computation**: Perform fast simulations, e.g., particle systems, numerical integration
- **Image Analysis**: Pixel-wise filters, transformations, masks
- **Feature Engineering**: Compute means, deltas, differences, or normalisations across axes

Pitfalls and Misconceptions

- **Shape mismatch in broadcasting**: Ensure trailing dimensions are compatible
- **Slicing returns views, not copies**: Use `.copy()` if needed
- **`reshape()` errors**: Total size must remain constant
- **Misuse of axis**: Confusing axis 0 vs 1 is a common source of bugs

NumPy Functions and Operations

NumPy Functions	Description
`np.array([1,2,3])`	Creates a 1D array of shape **1x3** with values *1, 2, 3*
`np.array([(1,2,3), (4,5,6)])`	Creates 2D array of shape **2x3** with values *1,2,3,4,5,6*
`np.array([[(1,2,3), (4,5,6)], [(7,8,9), (10,11,12)]])`	Creates a 3D array with shape **2x2x3**
`np.zeros(3,4)`	Creates a **3x4** array of zeros
`np.arrange(1,60,5)`	Creates a 1D array of values 1 through 60 at steps of 5
`arr.reshape(2,3,4)`	Reshapes array `arr` into an array of shape **2x3x4**
`arr.shape`	To get the shape of the array `arr`

NumPy Operations	Description
`a[0]`	Gets the 0th element in 1D array
`b[0,0]`	Gets the 0th element in 2D array
`c[0,0,0]`	Gets the 0th element in 3D array
`c[9,0,0]`	Gets the element which is the 0th element in the 0th row in the 9th depth
1D Arrays	
`a[:]`	Selects everything
`a[2:5]`	Selects the 2nd through the 4th rows (does not include the 5th row)
2D Arrays	

NumPy Operations	Description
`b[:,:]`	Selects all rows and all columns
`b[:,0]`	Selects all rows, and the zeroth column
`b[0,:]`	Selects the zeroth row, and all columns in that row
`b[0:2,:]`	Selects the zeroth and first row, but NOT the second row
`b[0:2,0:2]`	Selects the zeroth and first row, and the zeroth and first column
3D Arrays	
`c[:,:,:]`	Selects all rows and columns on all depths
`c[0,:,:]`	Selects the everything in the first depth
`c[:,0,:]`	Selects the first row of each depth
`c[:,:,0]`	Selects the first column of each depth

Try this out

1. Write NumPy code to create a 5×5 matrix with values 1 to 25.
2. Given `a = np.array([1, 2, 3])`, what is `a[:, np.newaxis] + a`?
3. How do you extract the diagonal of a 2D array?
4. How would you stack two 1D arrays column-wise?
5. What is the difference between `flatten()` and `ravel()`?

Additional Reading

- Official NumPy Docs
- Visual Guide to Broadcasting
- NumPy 100 Practice Problems

Pandas Series

Key Definitions & Concepts

`Series`: A one-dimensional labelled array, capable of holding any data type (int, float, str, datetime, etc.). Think of it as a hybrid of a list and a dictionary.

Index: The labels associated with each element in a Series. Allows fast access and alignment.

Vectorised Operations: Just like NumPy arrays, Series supports element-wise operations (addition, comparison, logical operations).

Method Chaining: The practice of applying multiple Series/DataFrame methods in a single statement. Promotes readable, fluent code.

`.str` **accessor**: Enables vectorised string methods.

`.dt` **accessor**: Provides datetime-specific operations after converting a Series to a datetime format.

Core Ideas

- A Pandas Series behaves like a **dictionary-backed vector**: values are like a NumPy array; labels (index) are like dictionary keys.
- Operations respect the index: this makes Pandas ideal for labelled time series or named features.
- **Method chaining** encourages writing "pipelines", stepwise transformations applied in sequence.
- Series enables **clean, readable, high-level operations** without the verbosity of loops.

Implementation

Creating Series

Python
```
import pandas as pd

# From a list
pd.Series([10, 20, 30])

# With custom index
pd.Series([10, 20, 30], index=['a', 'b', 'c'])

# From a dictionary
pd.Series({'x': 1, 'y': 2, 'z': 3})
```

Indexing and Slicing

Python
```
s = pd.Series([10, 20, 30], index=['a', 'b', 'c'])



s['a']             # 10

s[0:2]             # First two elements

s[['a', 'c']]      # Specific labels

s.loc['b']         # Access by label

s.iloc[1]          # Access by position
```

Operations on Series

```python
s = pd.Series([1, 2, 3, 4])

s + 10                    # [11, 12, 13, 14]
s * 2                     # [2, 4, 6, 8]
s.mean(), s.max(), s.std()

# Method chaining
s.add(1).mul(10).clip(0, 25)

# Remove duplicates
pd.Series([1, 2, 2, 3]).drop_duplicates()

# Apply/map
s.apply(lambda x: x**2)
s.map({1: 'A', 2: 'B', 3: 'C'})  # Mapping values
```

Working with Series (Boolean Logic & Filtering)

```python

s = pd.Series([100, 200, 300, 400], index=['a', 'b', 'c', 'd'])



s[s > 200]                # Filter by value

s.isnull()                # Check for missing values

s.fillna(0)               # Fill missing values
```

String Operations with `.str`

```python
names = pd.Series(['alice', 'BOB', 'Charlie'])


names.str.upper()                   # ['ALICE', 'BOB', 'CHARLIE']

names.str.len()                     # [5, 3, 7]

names.str.contains('li')           # [True, False, True]

names.str.split('a')                # [['', 'lice'], ['BOB'], ['Ch',
'rlie']]
```

Datetime Operations with `.dt`

```python
dates = pd.Series(['2023-01-01', '2023-05-15'])

dates = pd.to_datetime(dates)


dates.dt.year           # [2023, 2023]

dates.dt.month          # [1, 5]

dates.dt.weekday        # [6, 0]
```

Summary

Operation	Method	Description
Create Series	`pd.Series(data)`	From list, dict, array
Indexing	`s['a']`, `s.iloc[0]`	By label or position
Math ops	`s + 10`, `s * 2`	Element-wise operations
Filtering	`s[s > 100]`	Boolean condition
Map/Apply	`s.map()`, `s.apply()`	Transform values
String ops	`s.str.upper()`, `s.str.len()`	Vectorised string handling
Datetime ops	`s.dt.year`, `s.dt.month`	Extract date components

Real-world Use Cases

- **Feature Engineering**: Extract year/month from timestamps
- **Data Cleaning**: Lowercase names, split strings, remove duplicates
- **Time Series Analysis**: Weekly sales, monthly trends
- **Categorical Encoding**: Mapping string values to codes

Pitfalls and Misconceptions

- `.apply()` vs `.map()`:
  `map()` is for value substitution; `apply()` is more general (can apply any function)
- `.str` accessor only works on string-typed Series; convert if needed
- `to_datetime()` is required before using `.dt` accessor

Try this out

1. Create a Series of 5 cities with population values.
2. Convert `['2022-01-01', '2023-01-01']` into a datetime Series and extract the year.
3. How would you replace all lowercase names in a Series with uppercase versions?
4. Filter a Series of integers to only keep even numbers.
5. Explain the difference between `.apply()` and `.map()` with a small example.

Pandas DataFrames

Key Definitions & Concepts

`DataFrame`: A two-dimensional, size-mutable, labelled data structure with columns of potentially different types.

Indexing: Refers to selecting rows and columns using position (`iloc`) or labels (`loc`).

Fancy Indexing: Supplying a list or array of labels/positions to select specific rows or columns.

Method Chaining: Writing successive operations on a DataFrame in one statement.

Grouping: The act of splitting data into groups and applying an aggregation function.

Wrangling: The process of cleaning, transforming, and reshaping raw data into a usable format.

Core Ideas & Theoretical Intuition

- A DataFrame is like a **spreadsheet with labelled rows and columns**, built for high-performance operations.
- Think of `.iloc` as NumPy-style **position-based access**, and `.loc` as dictionary-style **label-based access**.
- Merging is analogous to SQL joins — critical for combining datasets from multiple sources.
- Grouping and pivoting allow **multilevel summarisation**, ideal for exploratory data analysis.
- Most DataFrame operations are **non-destructive**, enabling method chaining and functional transformations.

Implementation

Creating DataFrames

```python
import pandas as pd

# From dictionary
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)

# From list of lists
df2 = pd.DataFrame([[1, 2], [3, 4]], columns=['A', 'B'])

# With custom index
pd.DataFrame(data, index=['a', 'b'])
```

Attributes

```python
df.dtypes          # Data type of each column

df.ndim            # Number of dimensions

df.shape           # Rows × Columns

df.size            # Total elements

df.index           # Row index object

df.columns         # Column names

df.values          # Underlying NumPy array
```

Info & Summary Functions

```
Python

df.info()           # Structure summary

df.head(3)          # First 3 rows; similarly for df.tail(3)

df.describe()       # Stats for numeric columns
```

Reading Data (CSV Example)

```
Python

pd.read_csv('data.csv', header=0, index_col='ID',

            usecols=['ID', 'Name', 'Score'],

            nrows=100, skiprows=1)
```

Indexing and Slicing

`iloc` (position-based)

```
Python

df.iloc[0]                 # First row

df.iloc[:, 1]              # All rows, second column

df.iloc[0:2, 1:3]          # Rows 0-1, Columns 1-2
```

upGrad

loc (label-based)

```
Python
df.loc['a']                 # Row with label 'a'
df.loc[:, 'Age']            # Entire 'Age' column
df.loc['a':'b', ['Name']] # Slice rows and specific col
```

Fancy Indexing

```
Python
# Rows with labels 'a' and 'c' and columns 'Name' and 'Age'
df.loc[['a', 'c'], ['Name', 'Age']]
# Rows at positions 0 and 2 and columns at positions 0 and 1
df.iloc[[0, 2], [0, 1]]
```

Operations on DataFrames

```
Python
df * 10                    # Multiply numeric values
df['Bonus'] = df['Age'] * 0.1  # Add new column


df.sum(), df.mean()        # Aggregations
df.sort_values('Age')      # Sort rows
df.apply(len)              # Apply function to columns
```

```
df['Name'].map(str.upper)  # Element-wise string map


# Method chaining

df.dropna().query("Age > 20").sort_values('Age')
```

Working with DataFrames

```python
df.columns = df.columns.str.lower()       # Rename cols

df.rename(columns={'age': 'year'})         # Rename

df.fillna(0)                              # Fill missing
```

Merging DataFrames

```python
pd.merge(df1, df2, how='inner', on='ID')

# SQL-style merge

pd.concat([df1, df2], axis=0)

# Stack rows
```

- `how=` options: 'inner', 'left', 'right', 'outer'
- `on=` specifies column key; `axis=1` for column-wise merge

Analysing DataFrames

Python

```python
df.reset_index()                        # Move index to column

df['Score'].quantile(0.75)             # 75th percentile
```

Grouping and Aggregation

Python

```python
df.groupby('Department')['Salary'].sum()

df.groupby(['Gender','Department']).agg({'Age': 'mean', 'Salary':
'max'})
# Group the DataFrame by both 'Gender' and 'Department',
# then compute the mean age and max salary for each group
```

Pivoting

Python

```python
df.pivot(index='Date', columns='Region', values='Sales')



## Sample Output:

# Region      North  South

# Date

# 2023-01-01    100    150

# 2023-01-02    200    180
```

Wrangling a DataFrame

```python
df.drop_duplicates()                # drop duplicate rows

df.dropna()                         # drop rows with null values

df['City'] = df['City'].str.strip().str.title()

df['Date'] = pd.to_datetime(df['Date'])
```

Summary

Task	Example Code	Output Description
Row by label	`df.loc['a']`	Row with index label 'a'
Col by position	`df.iloc[:, 1]`	Second column
Filter	`df[df['Age'] > 25]`	Rows with Age > 25
Merge	`pd.merge(df1, df2, on='ID')`	Combine tables
Groupby	`df.groupby('Team')['Score'].mean()`	Aggregated stats
Pivot	`df.pivot(index, columns, values)`	Reshaped table

Pitfalls and Misconceptions

- `.iloc` vs `.loc`: Position vs Label: using the wrong one may raise errors
- Modifying views: `.loc` may return a view, not a copy: use `.copy()` if needed
- `map()` vs `apply()`: `map` works on Series elements; `apply` can be column-wise on DataFrame
- Be cautious when chaining inplace operations (e.g., `df.dropna(inplace=True)` followed by `df.sort_values()`)

Common Interview Questions

1. What are the key differences between a NumPy array and a Python list?

Topic: NumPy Basics
Expected Points:

- Fixed-size, homogeneous vs dynamic, heterogeneous
- Vectorised operations in NumPy
- Memory efficiency and performance benefits

2. Explain broadcasting in NumPy. Give an example of a valid and an invalid broadcast operation.

Topic: NumPy Broadcasting
Expected Points:

- Rules for shape compatibility
- Example: `(3, 1)` + `(1, 4)` → `(3, 4)`
- Invalid example where trailing dimensions don't match

3. What does `.reshape()` do in NumPy? How is it different from `.ravel()` and `.flatten()`?

Topic: Array Reshaping & Views
Expected Points:

- `reshape` changes shape without changing data
- `flatten` returns a copy; `ravel` returns a view (if possible)

4. How would you remove duplicate values from a Pandas Series or DataFrame?

Topic: Data Cleaning in Pandas
Expected Points:

- `Series.drop_duplicates()`
- `DataFrame.drop_duplicates(subset=['col1', ...])`

5. How are `.loc[]` and `.iloc[]` different? When would you use each?

Topic: Pandas Indexing
Expected Points:

- `.loc[]` uses label-based indexing
- `.iloc[]` uses integer position indexing
- Edge cases (e.g., mixed-type index)

6. What is method chaining in Pandas? Why is it useful?

Topic: Code Style and Efficiency
Expected Points:

- Writing operations like: `df.dropna().query(...).sort_values(...)`
- Benefits: readable, compact, avoids intermediate variables

7. Explain the difference between `map()`, `apply()`, and `applymap()` in Pandas.

Topic: Functional Programming with Pandas
Expected Points:

- `map()` → element-wise on Series
- `apply()` → row/column-wise on DataFrame
- `applymap()` → element-wise on DataFrame

8. Describe how you would merge two DataFrames in Pandas. What types of joins are supported?

Topic: Combining DataFrames
Expected Points:

- `pd.merge(left, right, how='inner', on='key')`
- `how=` supports: 'inner', 'left', 'right', 'outer'
- Can also join on the index or use `concat()` for stacking

9. You read a CSV file and find some columns are not needed, and the headers are in the second row. How would you load only the relevant data?

Topic: Data Import & Preprocessing
Expected Points:

- Use `usecols=`, `skiprows=`, `header=` in `read_csv()`

10. How would you calculate the average sales per product category in a DataFrame? How would you reshape the result into a pivot table?

Topic: Grouping and Pivoting

Expected Points:

- `df.groupby('Category')['Sales'].mean()`
- `df.pivot(index='Date', columns='Category', values='Sales')`