BASICS FOR MACHINE LEARNING

1. Basics of NumPy Arrays

NumPy arrays (ndarray) are the foundation of numerical computing in Python. Unlike Python lists, NumPy arrays are **homogeneous** and support **vectorized operations**, meaning computations happen faster using optimized C backend code.

Key Features:

- Fixed size and type (e.g., all elements must be integers, floats, etc.)
- Multi-dimensional support (1D, 2D, 3D…)
- Fast element-wise operations

Example:

```
import numpy as np

# 1D array
arr1 = np.array([1, 2, 3, 4, 5])

# 2D array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

print(arr1.shape)   # (5,)
print(arr2.shape)   # (2, 3)
```

Relation to Machine Learning :

In **Machine Learning**, NumPy arrays are used to represent datasets efficiently — for example, representing **features and samples** in machine learning models:

- Rows → individual samples
- Columns → features (like age, income, score, etc.)

2. Aggregations (Summarizing Data)

Aggregations compute **summary statistics** such as sum, mean, min, max, or standard deviation across an array or specific axes.

Common Aggregation Functions:

- np.sum(), np.mean(), np.std(), np.min(), np.max(), np.median(), np.var()

Example:

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

```
print(np.sum(data))       # 210
print(np.mean(data))      # 35.0
print(np.max(data, axis=0))  # [40, 50, 60]
print(np.min(data, axis=1))  # [10, 40]
```

Relation to Machine Learning :

Aggregation functions are used to get **insights from data**:

- Mean age of customers
- Standard deviation of prices
- Maximum sales in each region

They are the foundation of **exploratory data analysis (EDA)**.

3. Computations on Arrays (Vectorized Operations)

Vectorization means applying operations directly to arrays without explicit loops. NumPy executes these operations at **compiled C speed**.

Example:

```
arr = np.array([1, 2, 3, 4, 5])

# Element-wise arithmetic
print(arr + 10)  # [11 12 13 14 15]
print(arr * 2)   # [ 2  4  6  8 10]
print(arr ** 2)  # [ 1  4  9 16 25]
```

Relation to Machine Learning :

Vectorized operations allow for **fast feature transformations**, e.g.:

- Normalizing features: (x - mean) / std
- Scaling images or sensor data
- Performing linear algebra operations in ML algorithms (matrix multiplication)

4. Comparisons, Masks, and Boolean Logic

NumPy allows element-wise comparisons that return **Boolean arrays**. These arrays can be used to create **masks** (filters) to select or modify data.

Example:

```
data = np.array([10, 20, 30, 40, 50])

# Element-wise comparison
mask = data > 25
```

```
print(mask)          # [False False  True  True  True]

# Applying mask
filtered = data[mask]
print(filtered)      # [30 40 50]
```

Boolean logic:
You can combine multiple conditions using:

- & → and
- | → or
- ~ → not

Example:

```
mask = (data > 20) & (data < 50)
print(data[mask])  # [30 40]
```

Relation to Machine Learning :

Used for **data filtering** and **conditional selection**, for example:

- Select all rows where income > 50,000 and age < 30
- Filter outliers or missing data

5. Fancy Indexing

Fancy indexing allows you to access elements of an array using **integer arrays** or **lists of indices** rather than slices.

Example:

```
arr = np.array([100, 200, 300, 400, 500])

indices = [0, 2, 4]
print(arr[indices])  # [100 300 500]

# For 2D arrays
matrix = np.arange(12).reshape(3, 4)
print(matrix[[0, 2], [1, 3]])  # [ 1 11 ]
```

Relation to Machine Learning :

Fancy indexing is used to:

- Extract **specific samples or features** from datasets
- Shuffle or sample data randomly during training
- Perform **advanced selection** in multi-dimensional data (e.g., pixel extraction from images)

6. Structured Arrays

Structured arrays allow you to store **heterogeneous data** (like a table with columns of different data types) within a single NumPy array — similar to a database record or a Pandas DataFrame.

Example:

```
student_data = np.array([
    (1, 'Alice', 85.5),
    (2, 'Bob', 90.2),
    (3, 'Charlie', 78.9)
], dtype=[('id', 'i4'), ('name', 'U10'), ('score', 'f4')])

print(student_data['name'])   # ['Alice' 'Bob' 'Charlie']
print(student_data['score'] > 80)  # [ True  True False]
```

Relation to Machine Learning :

Structured arrays can represent **tabular datasets** before converting to Pandas DataFrames. They're useful when handling **sensor data**, **financial records**, or **experiment results** with multiple attributes.

Summary Table – NumPy Topics and Machine Learning Applications

NumPy Concept	Core Idea	Machine Learning Application
Arrays	Homogeneous, efficient data structures	Store numerical datasets efficiently
Aggregations	Summarize data	Compute mean, std, sum for analytics
Computations	Vectorized math operations	Feature scaling, transformations
Masks & Boolean Logic	Conditional filtering	Data cleaning, selecting subsets
Fancy Indexing	Advanced indexing using integer arrays	Sampling, feature extraction
Structured Arrays	Heterogeneous tabular data	Represent datasets with multiple attributes

Example: Machine Learning Workflow

```
import numpy as np

# Step 1: Create synthetic dataset
age = np.array([22, 35, 58, 45, 33, 26])
income = np.array([25000, 50000, 80000, 62000, 45000, 30000])

# Step 2: Aggregations
print("Average Income:", np.mean(income))

# Step 3: Computations (normalize)
income_norm = (income - np.min(income)) / (np.max(income) - np.min(income))
```

```
# Step 4: Comparisons and Masks (filter)
mask = (age < 40) & (income > 30000)
print("Selected incomes:", income[mask])

# Step 5: Fancy indexing (pick random 3 samples)
indices = np.random.choice(len(age), 3, replace=False)
print("Random sample ages:", age[indices])

# Step 6: Structured array
customers = np.array(list(zip(age, income)),
                     dtype=[('age', 'i4'), ('income', 'f4')])
print(customers['income'] > 50000)
```

This example simulates a small data analysis task — summarizing, normalizing, filtering, sampling, and structuring customer data — all essential operations in **data preprocessing** and **feature engineering** for Machine Learning .

1. Basics of Data Manipulation with Pandas

Pandas is a **Python library** built on top of NumPy for handling and analyzing **tabular and labeled data** efficiently. Its two core structures are:

- **Series** → 1D labeled array
- **DataFrame** → 2D labeled data (like a spreadsheet or SQL table)

Example:

```
import pandas as pd

# Creating a Series
s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])

# Creating a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 70000]
}
df = pd.DataFrame(data)
print(df)
```

Output:

```
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   35   70000
```

In Machine Learning :

DataFrames are the **foundation of data analysis**. They are used to:

- Load data from CSV, Excel, SQL, or APIs
- Clean, transform, and prepare data for modeling
- Perform Exploratory Data Analysis (EDA)

2. Data Indexing and Selection

Indexing helps access, filter, and modify subsets of data efficiently.

- .loc[] → label-based selection
- .iloc[] → integer-based selection
- Boolean indexing → conditional selection

Example:

```
print(df.loc[1])              # Select row with index label 1
print(df.iloc[0:2])           # Select first two rows
print(df[df['Salary'] > 55000]) # Conditional selection
```

In Machine Learning :

Used for **data exploration** and **feature selection**, e.g.:

- Selecting rows with missing values
- Extracting records for a particular group (e.g., customers above 40 years)

3. Operating on Data

Pandas allows **vectorized operations** and **functions** on entire columns or rows, similar to NumPy but with labeled data.

- Arithmetic: +, -, *, /
- Statistical: .mean(), .sum(), .std()
- Applying custom functions: .apply()

Example:

```
df['Bonus'] = df['Salary'] * 0.10
df['New_Salary'] = df['Salary'] + df['Bonus']

# Using apply()
df['Age_Group'] = df['Age'].apply(lambda x: 'Young' if x < 30 else 'Senior')
print(df)
```

In Machine Learning :

Used for **feature engineering** — creating new columns or features from existing data (e.g., normalized values, risk categories, etc.)

4. Handling Missing Data

Real-world data is often **incomplete or inconsistent**. Pandas provides tools to detect, remove, or fill missing values.

- df.isnull() → Detect missing values
- df.dropna() → Remove missing rows/columns
- df.fillna(value) → Replace missing values

Example:

```
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, None, 35],
        'Salary': [50000, 60000, None]}
df = pd.DataFrame(data)


print(df.isnull())        # Check missing values
df['Age'] = df['Age'].fillna(df['Age'].mean())  # Fill with mean
df = df.dropna(subset=['Salary'])               # Drop rows missing Salary
print(df)
```

In Machine Learning :

Cleaning missing data is part of **data preprocessing**, essential before model training to prevent bias and errors.

5. Hierarchical Indexing (MultiIndex)

Hierarchical or Multi-level indexing allows **multiple index levels** (like a composite key). Useful for representing **multi-dimensional** data compactly.

Example:

```
arrays = [
    ['India', 'India', 'USA', 'USA'],
    ['2023', '2024', '2023', '2024']
]
```

```
index = pd.MultiIndex.from_arrays(arrays, names=('Country', 'Year'))
data = pd.DataFrame({'GDP': [3.5, 3.7, 2.9, 3.0]}, index=index)
print(data)
```

Output:

Country	Year	GDP
India	2023	3.5
	2024	3.7
USA	2023	2.9
	2024	3.0

In Machine Learning :

Representing **panel data** (multi-dimensional time series)
Stock market data (Company × Year)
Organizing grouped results in analytics

6. Combining Datasets

Combining datasets is critical when merging data from multiple sources — like joining tables in SQL.

- pd.concat() → Stack datasets vertically or horizontally
- pd.merge() → SQL-style joins (inner, left, right, outer)
- df.join() → Simplified join on indices

Example:

```
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [1, 2, 3], 'Salary': [50000, 60000, 70000]})

merged = pd.merge(df1, df2, on='ID')
print(merged)
```

In Machine Learning :

Used for **data integration**:

- Combining **customer demographics** with **transaction data**
- Merging **training and testing datasets**
- Integrating **external APIs or CSV data sources**

7. Aggregation and Grouping

Grouping allows you to **split** data into groups, **apply** operations, and **combine** results — the **split-apply-combine** strategy.

- .groupby() for grouping data

- .agg() for applying multiple aggregation functions

Example:

```
data = {'Department': ['IT', 'HR', 'IT', 'HR', 'Sales'],
        'Salary': [60000, 50000, 65000, 48000, 55000]}
df = pd.DataFrame(data)

grouped = df.groupby('Department')['Salary'].mean()
print(grouped)

# Multiple aggregations
agg = df.groupby('Department')['Salary'].agg(['min', 'max', 'mean'])
print(agg)
```

Output:

```
Department
HR      49000.0
IT      62500.0
Sales   55000.0
```

In Machine Learning :

Used for **data summarization and feature extraction**, e.g.:

- Average sales by region
- Mean income per age group
- Aggregated metrics for model features

8. Pivot Tables

Pivot tables reshape data — similar to Excel — to summarize information using **grouping and aggregation**.

```
pd.pivot_table(data, values, index, columns, aggfunc)
```

Example:

```
data = {
    'Department': ['IT', 'HR', 'IT', 'HR', 'Sales'],
    'Year': [2023, 2023, 2024, 2024, 2024],
    'Salary': [60000, 50000, 65000, 48000, 55000]
}
df = pd.DataFrame(data)

pivot = pd.pivot_table(df, values='Salary', index='Department',
                       columns='Year', aggfunc='mean')
print(pivot)
```

Output:

Year 2023 2024
Department
HR 50000.0 48000.0
IT 60000.0 65000.0
Sales NaN 55000.0

In Machine Learning :

Pivot tables are used for:

- **Summarizing large datasets** for reporting
- Creating **feature tables** for ML (e.g., average purchase per year)
- Quick exploratory analysis

Pandas Topics and Machine Learning Applications

Concept	Core Idea	Machine Learning Relevance
Data Manipulation	Load, edit, and analyze tabular data	Foundation of all data analysis tasks
Indexing & Selection	Accessing subsets of data	Data filtering and extraction
Operating on Data	Vectorized operations on columns	Feature engineering
Missing Data	Handle null or incomplete values	Data cleaning and preprocessing
Hierarchical Indexing	Multi-level index	Multi-dimensional analytics
Combining Datasets	Merge/join multiple datasets	Data integration from various sources
Aggregation & Grouping	Summarize groups	EDA and statistical summaries
Pivot Tables	Reshape and summarize data	Business analytics and visualization

Example Code

```
import pandas as pd

# Step 1: Load sample data
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Department': ['IT', 'HR', 'IT', 'HR', 'Sales'],
    'Salary': [60000, 50000, 65000, None, 55000],
    'Experience': [2, 5, 7, 3, 4]
}
df = pd.DataFrame(data)

# Step 2: Handle missing data
df['Salary'].fillna(df['Salary'].mean(), inplace=True)

# Step 3: Feature engineering
df['Bonus'] = df['Salary'] * 0.1
df['Level'] = df['Experience'].apply(lambda x: 'Junior' if x < 5 else 'Senior')
```

```
# Step 4: Grouping and aggregation
summary = df.groupby('Department')['Salary'].agg(['mean', 'max', 'count'])

# Step 5: Pivot for reporting
pivot = pd.pivot_table(df, values='Salary', index='Level', columns='Department')

print(summary)
print(pivot)
```

This example simulates a **complete data manipulation workflow** — cleaning, transformation, grouping, and summarization — all key steps before building **data science or machine learning models**.