Data Analysis Techniques in Python

Setting Up Python Libraries in VS Code

Install Libraries:

python -m pip install numpy pandas matplotlib

Test Installation:

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

Libraries are pre-written collections of code that provide ready-made functions, classes, and tools to perform specific tasks, so programmers don't have to write everything from scratch.

They help to:

- Save time
- Reduce errors
- Make programs efficient and readable

Examples

- NumPy → numerical computations using arrays
- Pandas → data analysis and manipulation
- Matplotlib → data visualization and plotting

NumPy & n-Dimensional Arrays

In food technology, data often comes from:

- Moisture analysis
- Texture profile analysis
- Shelf-life studies
- Sensory scores
- Replicated experiments

NumPy arrays are used to store such numerical data efficiently.

Key concepts:

- 1D array → single experiment readings
- 2D array → multiple samples × multiple parameters

Moisture content (%) of a food sample measured 5 times:

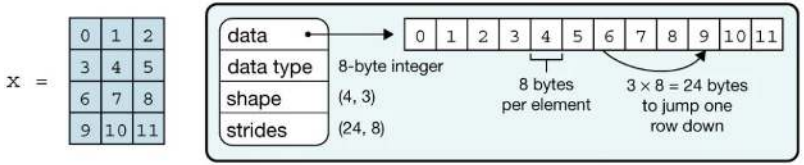
```
import numpy as np

moisture = np.array([12.5, 12.8, 12.6, 12.7, 12.9])

print("Average moisture:", moisture.mean())
```

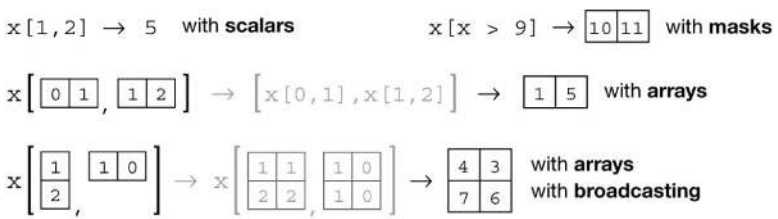
NumPy & n-Dimensional Arrays

a Data structure

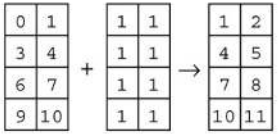


b Indexing (view)

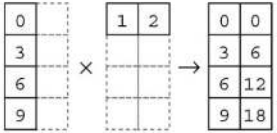
c Indexing (copy)



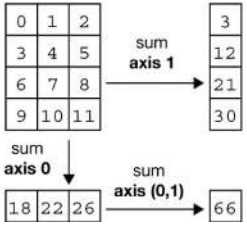
d Vectorization



e Broadcasting



f Reduction



g Example

```
In [1]: import numpy as np

In [2]: x = np.arange(12)

In [3]: x = x.reshape(4, 3)

In [4]: x
Out[4]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

In [5]: np.mean(x, axis=0)
Out[5]: array([4.5, 5.5, 6.5])

In [6]: x = x - np.mean(x, axis=0)

In [7]: x
Out[7]:
array([[-4.5, -4.5, -4.5],
       [-1.5, -1.5, -1.5],
       [ 1.5,  1.5,  1.5],
       [ 4.5,  4.5,  4.5]])
```

NumPy & n-Dimensional Arrays

A NumPy array is a homogeneous, fixed-size, multidimensional array stored in contiguous memory.

Key components:

- Data: actual elements stored sequentially
- Data type (dtype): type of each element (e.g., int64 → 8 bytes)
- Shape: dimensions of the array (rows, columns)
- Strides: number of bytes to move to the next element in each dimension

This structure allows fast computation compared to Python lists.

NumPy & n-Dimensional Arrays

```
import numpy as np              # Import NumPy library

x = np.arange(12)               # Create a 1D array with values 0 to 11

print(x)

x = x.reshape(4, 3)             # Reshape array into 4 rows and 3 columns

print(x)                        # Display the array

print("Shape:", x.shape)        # Print dimensions of the array (rows, columns)

print("Data type:", x.dtype)    # Print data type of elements

print("Strides:", x.strides)    # Print memory step size (bytes) for each axis
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
Shape: (4, 3)
Data type: int64
Strides: (24, 8)
```

NumPy & n-Dimensional Arrays

```
[[0]
 [3]
 [6]
 [9]]
[[ 0  2]
 [ 3  5]
 [ 6  8]
 [ 9 11]]
```

Indexing (view-slicing)

- Slicing returns a view, not a copy
- Views share the same memory as the original array
- Slice format: `start : end : step`

Changes in the view affect the original array.

```
col = x[:, :1]        # Select all rows and first column (view, not copy)

step_slice = x[:, ::2]          # Select all rows, every 2nd column using step

print(col)                      # Print column slice
print(step_slice)               # Print stepped slice
```

NumPy & n-Dimensional Arrays

```
5
[10 11]
[1 5]
```

Indexing (copy)

- Scalar indexing returns a single value
- Boolean masking creates a copy
- Fancy indexing (using arrays) also creates a copy

These do not share memory with the original array.

```
value = x[1, 2]                 # Access element at row 1, column 2 (scalar)

masked = x[x > 9]               # Boolean masking: select elements greater than 9

fancy = x[[0, 1], [1, 2]]       # Fancy indexing using index arrays

print(value)                    # Print scalar value
print(masked)                   # Print masked array
print(fancy)                    # Print fancy indexed result
```

NumPy & n-Dimensional Arrays

Vectorization

Vectorization means performing operations on entire arrays at once, without explicit loops.

It is:

- Faster
- Cleaner
- More memory-efficient

```
[[1 1]
 [1 1]
 [1 1]
 [1 1]]
```

```
[[ 1  2]
 [ 4  5]
 [ 7  8]
 [10 11]]
```

```
a = np.array([[0, 1],            # Create a 2D array
              [3, 4],
              [6, 7],
              [9, 10]])

b = np.ones((4, 2), dtype=int)   # Create a 4x2 array filled with ones
print(b)
result = a + b                  # Add arrays element-wise (vectorized operation)

print(result)                   # Print result of vectorized addition
```

NumPy & n-Dimensional Arrays

Broadcasting

Broadcasting allows NumPy to perform operations on arrays of different shapes by automatically expanding dimensions.
Rules (simplified):

1. Dimensions must be equal or
2. One of them must be 1

```
col = x[:, 0]                    # Extract first column as a 1D array
print(col)
row = np.array([1, 2, 3])        # Create a 1D row array
print(row)
col = col.reshape(-1, 1)         # Reshape column to 2D for broadcasting
print(col)
result = col * row               # Multiply using broadcasting rules

print(result)                    # Print broadcasted multiplication result
```

```
[0 3 6 9]
```

```
[1 2 3]
```

```
[[0]
 [3]
 [6]
 [9]]
```

```
[[ 0  0  0]
 [ 3  6  9]
 [ 6 12 18]
 [ 9 18 27]]
```

NumPy & n-Dimensional Arrays

Reduction

Reduction operations collapse dimensions by applying an operation along an axis.

Common reductions:

- `sum`
- `mean`
- `max`
- `min`

Axis meaning:

- `axis=0` → column-wise
- `axis=1` → row-wise

NumPy & n-Dimensional Arrays

Reduction

```
row_sum = np.sum(x, axis=1)      # Sum elements across columns (row-wise)

col_sum = np.sum(x, axis=0)      # Sum elements across rows (column-wise)

total_sum = np.sum(x, axis=(0, 1))  # Sum all elements in the array

print(row_sum)                   # Print row-wise sum
print(col_sum)                   # Print column-wise sum
print(total_sum)                 # Print total sum
```

```
[ 3 12 21 30]
[18 22 26]
66
```

NumPy & n-Dimensional Arrays

Reduction

```
row_sum = np.sum(x, axis=1)      # Sum elements across columns (row-wise)

col_sum = np.sum(x, axis=0)      # Sum elements across rows (column-wise)

total_sum = np.sum(x, axis=(0, 1))  # Sum all elements in the array

print(row_sum)                   # Print row-wise sum
print(col_sum)                   # Print column-wise sum
print(total_sum)                 # Print total sum
```

```
[ 3 12 21 30]
[18 22 26]
66
```

NumPy & n-Dimensional Arrays

Reduction

```
row_mean = np.mean(x, axis=1)          # Calculate mean of each row

col_mean = np.mean(x, axis=0)          # Calculate mean of each column

total_mean = np.mean(x)                # Calculate mean of all elements

print(row_mean)                        # Print row-wise mean
print(col_mean)                        # Print column-wise mean
print(total_mean)                      # Print overall mean
```

```
[ 1.  4.  7. 10.]
[4.5 5.5 6.5]
5.5
```

NumPy & n-Dimensional Arrays

Reduction

```
row_max = np.max(x, axis=1)                    # Find maximum value in each row

col_max = np.max(x, axis=0)                    # Find maximum value in each column

total_max = np.max(x)                          # Find maximum value in the array

print(row_max)                                 # Print row-wise maximum
print(col_max)                                 # Print column-wise maximum
print(total_max)                               # Print overall maximum
```

```
[ 2  5  8 11]
[ 9 10 11]
11
```

NumPy & n-Dimensional Arrays

Reduction

```
row_min = np.min(x, axis=1)                    # Find minimum value in each row

col_min = np.min(x, axis=0)                    # Find minimum value in each column

total_min = np.min(x)                          # Find minimum value in the array

print(row_min)                                 # Print row-wise minimum
print(col_min)                                 # Print column-wise minimum
print(total_min)                               # Print overall minimum
```

```
[0 3 6 9]
[0 1 2]
0
```

Pandas DataFrames

What is a DataFrame?

A Pandas DataFrame is a 2-dimensional, table-like data structure similar to an Excel sheet.

In food experiments:

- Rows represent *samples* (raw materials, batches, or treatments)
- Columns represent *measured parameters* such as:
    - Moisture
    - Protein
    - Fat
    - pH

Why DataFrames are important in food analysis

They allow food scientists to:

- Store experimental results in a structured form
- Filter samples that meet or fail quality standards
- Summarize nutritional or physicochemical data
- Export results for reports, audits, or regulatory submission

Pandas DataFrames

Food experiments generate **tabular data**:

- Rows → samples
- Columns → parameters (pH, moisture, fat, protein)

Pandas makes it easy to:

- Store
- Filter
- Summarize
- Export results

```
import pandas as pd  # Import pandas library

# Experimental data from food samples
data = {
    "Sample": ["S1", "S2", "S3"],        # Sample identifiers
    "Moisture": [12.5, 13.1, 12.8],      # Moisture content (%)
    "Protein": [8.2, 8.0, 8.5]           # Protein content (%)
}

# Create DataFrame
#pd.DataFrame()converts raw experimental values into an analyzable table
df = pd.DataFrame(data)

# Display the DataFrame
print(df)
```

```
  Sample  Moisture  Protein
0     S1      12.5      8.2
1     S2      13.1      8.0
2     S3      12.8      8.5
```

Pandas DataFrames

Reading & Writing CSV / Excel File

Why file handling matters in food labs

Food industry and research labs frequently:

- Import raw data from instruments (CSV files)
- Share results with QA teams (Excel files)
- Archive batch data for traceability

```
import pandas as pd
# Read experimental data from CSV file
df = pd.read_csv("foodData.csv")
print(df.head())
```

Pandas supports CSV and Excel, the most common formats in food science.

For Excel file support:

pip3 install openpyxl

	Sample_ID	Storage_Days	Moisture_Content_%	pH	Water_Activity_aw
0	1	19	12.51	7.05	0.608
1	2	23	11.62	6.46	0.624
2	3	60	12.73	6.94	0.655
3	4	12	13.11	6.36	0.662
4	5	40	13.19	6.54	0.662

Pandas DataFrames

Reading & Writing CSV / Excel File

Use `to_csv()` to write data to a `.csv` file.
Use `to_excel()` to write data to a `.xlsx` file.

For Excel file support:

pip3 install openpyxl

```
import pandas as pd
# Read experimental data from CSV file
df = pd.read_csv("foodData.csv")
first_five_row=df.head()
print(first_five_row)
```

```
#export
# Export results to Excel for reporting
first_five_row.to_csv("food_results.csv",
index=False)
```