Data Analytics & Statistics in Python
Session 2: Data Frames

Metropolia
University of Applied Sciences

Learning data-driven decision-making with Python

Instructor: Hamed Ahmadinia, Ph.D.

Email: hamed.ahmadinia@metropolia.fi

Concepts of Today

- **Data Frames & Matrix:**
  - **Arrays and Matrices:** Organized data in rows and columns for easy calculations.
  - **NumPy Basics and Advanced Operations:** Faster tools for working with large numbers and reshaping data.
  - **Pandas Series and DataFrames:** Tables with labels to organize and work with data easily.
  - **Data Editing and Filtering:** Clean and filter data to focus on what's important.
  - **Grouping, Merging, and Concatenating Data:** Combine or group data for summaries and comparisons.
  - **Time-Series Data:** Data linked to dates and times for tracking changes over time

Arrays and NumPy Overview

- **What are Arrays?**
  - Arrays store collections of **numerical elements** of the **same type**.
  - Data is stored **contiguously in memory** for fast access.
  - **Efficient for numerical operations** like addition and multiplication.

```
Array: [1, 2, 3, 4]

Operation: Add 2 → [3, 4, 5, 6]
```

- **What is NumPy?**
  - **NumPy:** A Python library for working with arrays and matrices.
  - **Why use NumPy?**
    - **Faster** than Python lists (optimized with C-based operations).
    - Supports **multi-dimensional arrays** (like matrices).
    - Built-in **math operations**: Addition, multiplication, etc.

```
List: [1, 2] + [3, 4] → Error!

NumPy Array: [1, 2] + [3, 4] → [4, 6]
```

Creating Arrays in NumPy

Table 1 : Creating Arrays: Common Methods

Method	Description	Example Output
np.array ([1, 2, 3])	Converts a list to an array	[1, 2, 3]
np.arrange (0, 10, 2)	Creates evenly spaced values (start, stop, step)	[0, 2, 4, 6, 8]
np.linspace (0, 1, 5)	5 evenly spaced values between 0 and 1	[0. , 0.25, 0.5, 0.75, 1.]
np.zeros ((3, 3))	3x3 matrix of zeros	[[0. 0. 0.], [0. 0. 0.], [0. 0. 0.]]
np.ones ((2, 4))	2x4 matrix of ones	[[1. 1. 1. 1.], [1. 1. 1. 1.]]
np.eye (3)	3x3 identity matrix	[[1. 0. 0.], [0. 1. 0.], [0. 0. 1.]]

List to Array

```
import numpy as np
arr = np.array([1, 2, 3])
print(arr)  # Output: [1 2 3]
```

Even Spaced Values

```
arange_arr = np.arange(0, 10, 2)
print(arange_arr)  # Output: [0 2 4 6 8]
```

Reshaping Arrays in NumPy

Table 2: Reshaping Arrays: Key Functions

Function	Description	Example Output
reshape()	Reshape array (e.g., 1D to 2D)	Reshapes [1, 2, 3, 4, 5, 6] → [[1, 2, 3], [4, 5, 6]]
ravel ()	Flatten multi-dimensional array to 1D	Flattens [[1, 2], [3, 4]] → [1, 2, 3, 4]
T (transpose)	Swap rows and columns	Transposes [[1, 2], [3, 4]] → [[1, 3], [2, 4]]
newaxis	Add new axis for reshaping	Adds dimension to [1, 2, 3] → [[1, 2, 3]]

Reshape 1D to 2D

```
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped = arr.reshape(2, 3)
print(reshaped)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

Flatten Array

```
arr2d = np.array([[1, 2], [3, 4]])
flat = arr2d.ravel()
print(flat)  # Output: [1 2 3 4]
```

Transpose Array

```
transposed = arr2d.T
print(transposed)
# Output:
# [[1 3]
#  [2 4]]
```

NumPy Overview and Special Values

- **Why Use NumPy?**
  - Handles large datasets efficiently.
  - Performs fast numerical calculations (e.g., addition, multiplication).
  - Widely used for scientific and machine learning tasks.

Special Values in NumPy

1. **NaN (Not a Number)** → Represents missing or invalid data
2. **Inf and -Inf (Infinity)** → Very large or small values.
3. **Use np.isinf (arr)** to check for infinity

```
Array: [NaN, 0, 1] → Mask: [True, False, False] → Filtered: [NaN]
```

```python
a = np.inf

print(a + 4)  # Output: inf
```

Combining and Sorting Arrays

- **Concatenating Arrays (Joining Arrays)**
  - np.concatenate ((a, b), axis=0) → Joins arrays along rows or columns.
  - np.vstack ((a, b)) → Stacks vertically (one below the other)
  - np.hstack ((a, b)) → Stacks horizontally (side by side).

```
a = [1, 2, 3]
b = [4, 5, 6]
Result: [1, 2, 3, 4, 5, 6]
```

- **Sorting Arrays (Rearranging Data)**
  - np.sort(arr) → Sorts values in the array.
  - np.argsort(arr) → Returns positions of sorted values (indices).

```
arr = np.array([3, 1, 2])
print(np.sort(arr))  # Output: [1 2 3]
```

Array Masking (Filtering Data)

- **What is Array Masking?**
  - A way to select elements of an array based conditions.
  - Uses boolean arrays (True/False) to filter da

```
arr = np.array([1, 2, 3, 4])
mask = arr > 2  # [False, False, True, True]
filtered_arr = arr[mask]  # Output: [3, 4]
```

- **Logical Operators for Combining Conditions:**
  - & (AND): Both conditions must be True.
  - | (OR): At least one condition must be True.
  - ~ (NOT): Inverts True to False and vice versa.

```
mask = (arr > 2) & (arr < 4)  # Output: [3]
```

Linear Algebra with NumPy

- **Key Linear Algebra Operations**
  - np.dot(a, b) → Dot Product (Vector Multiplication): Multiplies and sums corresponding elements
  - np.linalg.inv(M) → Returns the inverse of a matrix
  - A @ B → Matrix Multiplication

Multiplies two matrices

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
result = A @ B  # Output: [[19, 22], [43, 50]]
```

Vector Multiplication

```
a = [1, 2]
b = [3, 4]
dot_product = np.dot(a, b)  # (1*3) + (2*4) = 11
print(dot_product)  # Output: 11
```

Inverse of a Matrix

```
M = np.array([[1, 2], [3, 4]]
M_inv = np.linalg.inv(M)
print(M_inv)
# Output:
# [[-2.   1. ]
#  [ 1.5 -0.5]]
```

Pandas – Series and DataFrames

- **Why Pandas?**
  - Makes working with large datasets easier.
  - Supports filtering, grouping, and statistical operations.
  - Integrates well with visualization libraries (e.g., Matplotlib).
- **Series (1D Labeled Arrays)**
  - A Series is like a one-dimensional array, but with labels (index) for each value.
  - Useful for representing single columns of data with meaningful labels.
- **DataFrame (2D Table)**
  - A DataFrame is like a table, with rows and columns.
  - Each column is a Series and can have different data types.

```
pd.Series([4, 5, 3], index=["Jack", "Mathew", "Connor"])
```

```
pd.DataFrame({"col1": [1, 2], "col2": [3, 4]})
```

Selecting and Editing Data

- **Selecting Data**
  - Selecting columns: Choose one or more columns by their names
  - Selecting rows: Slice rows using their position or labels.
- **Indexing Methods**
  - iloc[] is great when you need specific rows/columns by position
  - loc[] is useful when working with labeled data, such as dates or names.
- **Editing Data**
  - You can modify specific values or entire sections of the DataFrame.
  - New columns can be added easily by assigning values directly.

```
# Sample DataFrame
df = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Age': [25, 30], 'Score': [85, 90]})

# 1. Selecting Data
print(df['Name'])  # Single column
print(df[0:1])  # First row

# 2. Indexing Methods
print(df.iloc[0, 2])  # Position-based: 1st row, 3rd column
print(df.loc[0, 'Score'])  # Label-based: row 0, column 'Score'

# 3. Editing Data
df.loc[0, 'Score'] = 95  # Update value
df['Status'] = ['Pass', 'Fail']  # New column
print(df)
```

Data Handling Methods in Pandas

Table 3: Data Handling and Transformation

Method	Purpose	Example
dropna()	Remove rows/columns with missing values	df.dropna(axis=0, how='any')
fillna (value)	Fill missing values with a specific value	df.fillna(0)
isna()	Identify missing values	df.isna()
apply(function)	Apply a function to columns/rows	df['Square'] = df['Age'].apply(lambda x: x ** 2)
map(mapping)	Map specific values in a column	df['Group'] = df['Age'].map({25: 'Young', 30: 'Adult'})
rename()	Rename columns	df.rename(columns={'Name': 'FullName'})
astype(dtype)	Change data type of a column	df['Age'] = df['Age'].astype(float)

Example of Data Handling and Transformation

```
# Sample DataFrame
df = pd.DataFrame({'Name': ['Alice', 'Bob', None], 'Age': [25, None, 30]})

# Handling Missing Data
df_cleaned = df.dropna()  # Removes rows with missing values
df_filled = df.fillna(0)  # Fill missing values with 0

# Adding and Modifying Columns
df['Status'] = ['Active', 'Inactive', 'Pending']  # New column
df['Square'] = df['Age'].apply(lambda x: x ** 2)  # Age squared

# Value Mapping and Renaming
df['AgeGroup'] = df['Age'].map({25: 'Young', 30: 'Adult'})  # Map age groups
df = df.rename(columns={'Name': 'FullName'})  # Rename column
df['Age'] = df['Age'].astype(float)  # Change data type to float
print(df)
```

Grouping Data in Pandas

- **Grouping Data**
  - Purpose: Group rows based on one or more columns and apply aggregation functions (e.g., mean(), sum()).
  - Method: groupby()

```python
import pandas as pd

# Creating a sample DataFrame
df = pd.DataFrame({
    'Category1': ['A', 'B', 'A', 'B', 'A', 'B'],
    'Value1': [1, 2, 3, 4, 5, 6],
    'Value2': [10, 20, 30, 40, 50, 60]
})

# Group by "Category1" and calculate the mean of "Value1"
groups = df.groupby("Category1")
print(groups["Value1"].mean())  # Mean of Value1 for each group
```

```
Category1
A    3.0
B    4.0
Name: Value1, dtype: float64
```

Time-Series Data

- **Concatenating DataFrames**
  - Purpose: Combine DataFrames vertically (rows) or horizontally (columns).
  - Method: pd.concat()

Parameter	Description
objs	List of DataFrames to combine
axis	0 for rows (default), 1 for columns
join	outer (union), inner (intersection)
ignore_ind	Reassign row index if True

```python
# Sample DataFrames
df1 = pd.DataFrame({'id': [1, 2], 'name': ['Alice', 'Bob'], 'age': [25, 30]})
df2 = pd.DataFrame({'id': [3, 4], 'name': ['Charlie', 'David'], 'age': [35, 40]})

# Vertical Concatenation (add rows)
concat_vertical = pd.concat([df1, df2], axis=0, ignore_index=True)
print("Vertical Concatenation:")
print(concat_vertical)

# Horizontal Concatenation (add columns)
df3 = pd.DataFrame({'city': ['New York', 'London'], 'salary': [50000, 60000]})
concat_horizontal = pd.concat([df1, df3], axis=1)
print("\nHorizontal Concatenation:")
print(concat_horizontal)
```

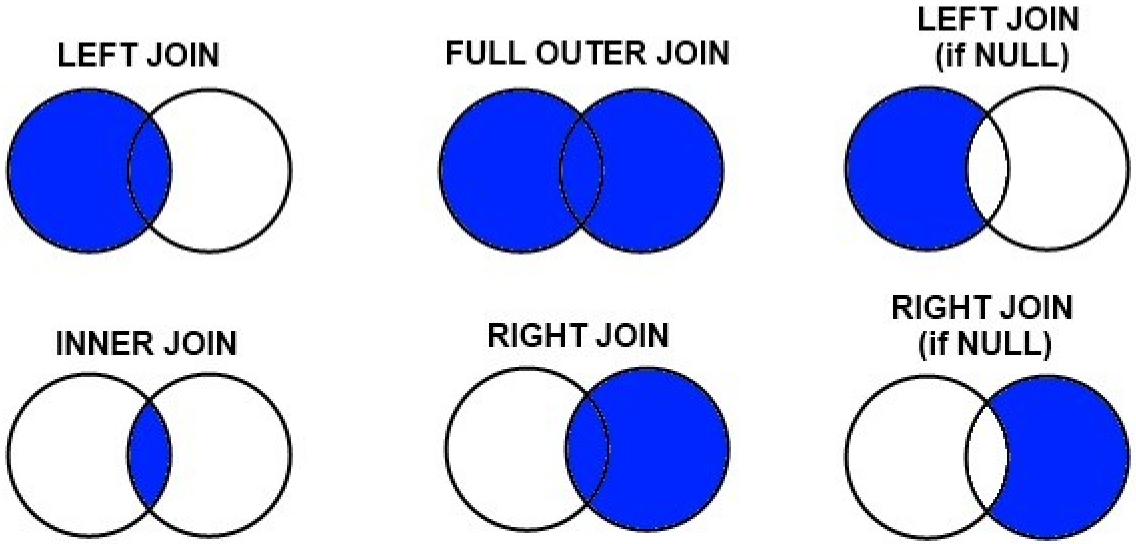
```
Vertical Concatenation:
   id     name  age
0   1    Alice   25
1   2      Bob   30
2   3  Charlie   35
3   4    David   40

Horizontal Concatenation:
   id   name  age      city  salary
0   1  Alice   25  New York   50000
1   2    Bob   30    London   60000
```

Merging DataFrames in Pandas

- **Merging Data**
  - Purpose: Combine DataFrames based on common columns or indices.
  - Method: merge()

Parameter	Description
on	Column name to merge on
how	Type of merge (left, right, inner, outer)
left_on/ right_on	Specify columns if names differ



Example for Types of Merges

```python
# Sample DataFrames
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Anna', 'Ben', 'Cara']})
df2 = pd.DataFrame({'ID': [2, 3, 4], 'Sport': ['Football', 'Basketball', 'Tennis']})

# Merges
print("Inner Merge:\n", df1.merge(df2, on='ID', how='inner'))
print("\nLeft Merge:\n", df1.merge(df2, on='ID', how='left'))
print("\nOuter Merge:\n", df1.merge(df2, on='ID', how='outer'))
```

```
Inner Merge:
    ID  Name       Sport
0   2   Ben    Football
1   3  Cara  Basketball

Left Merge:
    ID  Name       Sport
0   1  Anna         NaN
1   2   Ben    Football
2   3  Cara  Basketball

Outer Merge:
    ID  Name       Sport
0   1  Anna         NaN
1   2   Ben    Football
2   3  Cara  Basketball
3   4   NaN      Tennis
```

Time-Series Data in Pandas

- **Key Concepts:**
  - Datetime Objects: Represent dates and times with properties like year, month, day, hour, etc.
  - Time-Series Data: Data indexed by timestamps, commonly used in financial and weather datasets.
- **Creating Time Series:**
  - Converting to Datetime: pd.to_datetime() converts date strings or columns into datetime format.

```
# Sample data
df = pd.DataFrame({'year': [2023, 2024, 2025],
                   'month': [8, 5, 1],
                   'day': [15, 10, 8]})

# Convert to datetime and set as index
df['date'] = pd.to_datetime(df[['year', 'month', 'day']])
df.set_index('date', inplace=True)
print(df)
```

```
            year  month  day
date
2023-08-15  2023      8   15
2024-05-10  2024      5   10
2025-01-08  2025      1    8
```

Time-Series Operations in Pandas

- **Key Operations:**
  - **Indexing and Slicing:** Select data for specific date ranges using df.loc['start_date':'end_date’].
  - **Date Arithmetic:** Subtract dates to calculate time differences (e.g., days between events).
  - **Resampling: Adjust time-series frequency:**
    - Upsampling (Daily): df.resample('D').mean()
    - Downsampling (Monthly): df.resample(‘ME').sum()

```python
# Sample DataFrame with datetime index
df = pd.DataFrame({'value': [100, 200, 300]},
                  index=pd.to_datetime(['2022-01-01', '2022-02-01', '2022-03-01']))

# Time difference since first date
df['time_from_start'] = df.index - df.index[0]

# Resampling to monthly means
monthly_mean = df.resample('ME').mean()

print("Time Difference:\n", df)
print("\nMonthly Mean:\n", monthly_mean)
```

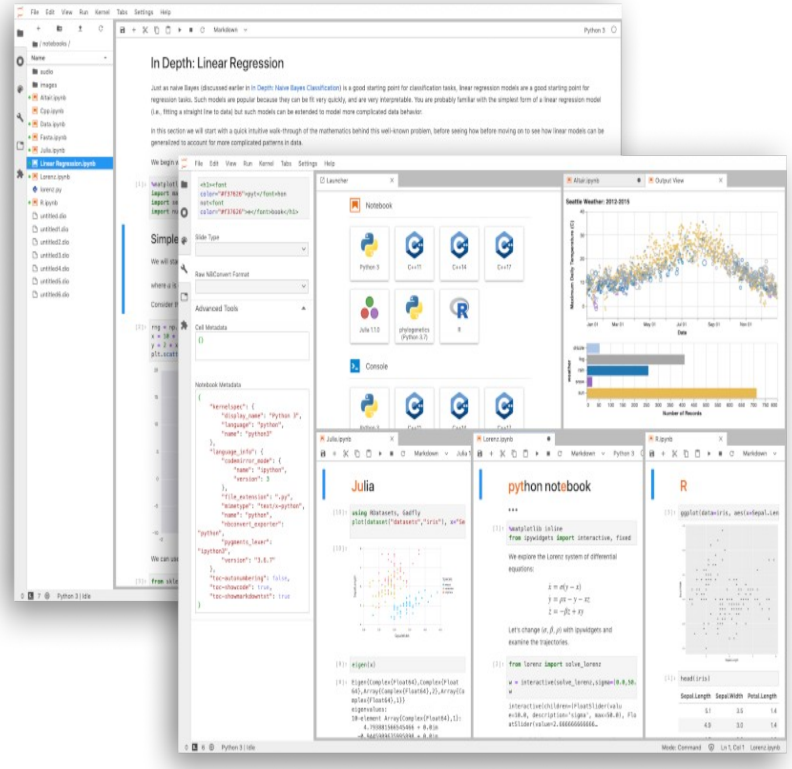
```
Time Difference:
            value time_from_start
2022-01-01    100          0 days
2022-02-01    200         31 days
2022-03-01    300         59 days

Monthly Mean:
            value time_from_start
2022-01-31  100.0          0 days
2022-02-28  200.0         31 days
2022-03-31  300.0         59 days
```

Notebook Review

Walk through how to apply key Python concepts in a Jupyter Notebook:

- Arrays and Matrices
- NumPy Basics and Advanced Operations
- Pandas Series and DataFrames
- Data Editing and Filtering
- Grouping, Merging, and Concatenating Data
- Time-Series Data



Kahoot Quiz Time!

Let’s Test Our Knowledge!

Hands-on Exercise

Form groups (2–3 members).

- Download *Hands-on Exercise #2* from the course page.
- Complete the coding tasks and discuss your solutions.
- Don’t forget to add the names of your group members to the file.
- Submit your completed *Hands-on Exercise* to the course Moodle page or send it to the teacher's email address.

Reference

- Vohra, M., & Patil, B. (2021). A Walk Through the World of Data Analytics. , 19-27. https://doi.org/10.4018/978-1-7998-3053-5.ch002.
- VanderPlas, J. (2016). Python data science handbook: Essential tools for working with data. O'Reilly Media. Available at https://jakevdp.github.io/PythonDataScienceHandbook/
- Severance, C. (2016). Python for everybody: Exploring data using Python 3. Charles Severance. Available at https://www.py4e.com/html3/
- McKinney, W. (2017). *Python for data analysis: Data wrangling with pandas, NumPy, and Jupyter.* O'Reilly Media.