جامعة الامير سلطان
PRINCE SULTAN UNIVERSITY

FOR

SPECIALIZATION

Robotics & Internet-of-Things

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
Pandas Data Structures

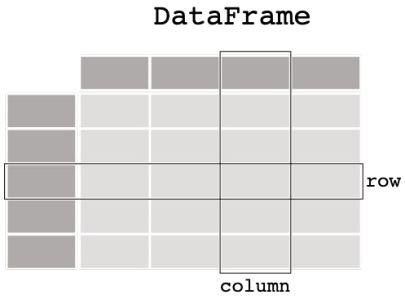
LECTURE 1
Pandas For Data Analytics
Create, Access, Load and Export Data

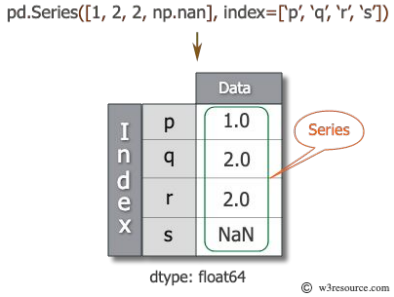
Prof. Anis Koubaa

SEP 2024

What is Pandas?

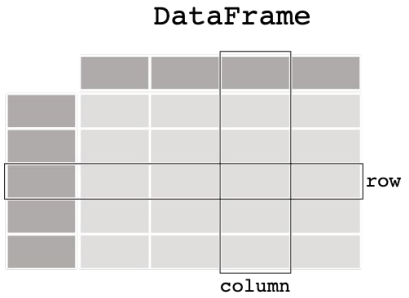
- **Pandas Overview**
  - Python library for **data manipulation** and **analysis**.
  - Built on **NumPy**, leveraging high-performance arrays.
- **Why Pandas?**
  - **Efficient** for working with structured data.
  - Simplifies tasks like data cleaning, exploration, and transformation.
  - Suitable for small to large datasets.

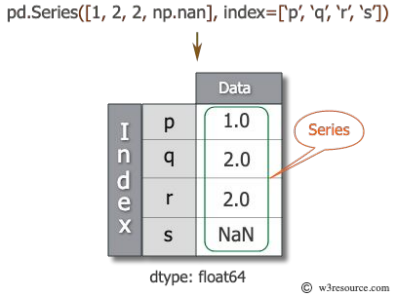




Why Use Pandas?

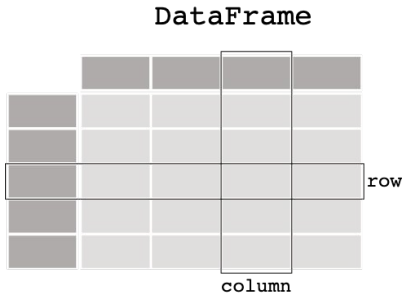
- **Advantages of Pandas**
  - **Simplifies complex tasks** like merging, reshaping, or filtering data.
  - **High flexibility**, handles missing data, time series, etc.
  - Supports **custom data transformations** using lambda functions.
- **Powerful I/O Tools**
  - Easily reads/writes to formats like **CSV**, **Excel**, **SQL**, and **JSON**.
  - Handles **large datasets** efficiently with chunking and memory optimization.

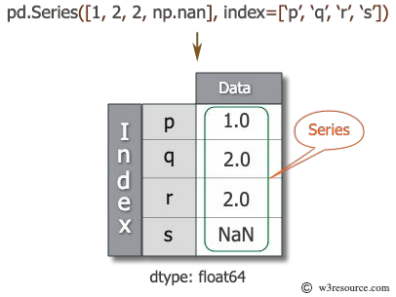




Key Features of Pandas

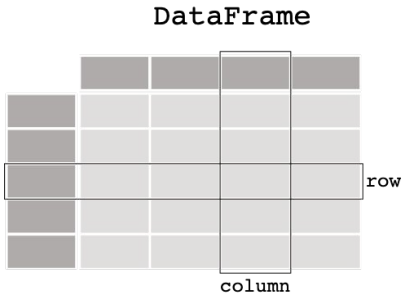
- **Core Data Structures**
  - **Series**: One-dimensional labeled array.
  - **DataFrame**: Two-dimensional table with labeled axes (rows, columns).
- **Key Functionalities**
  - Handling **missing data** with `.isnull()`, `.dropna()`, `.fillna()`.
  - Flexible **grouping and aggregation** with `.groupby()`, `.agg()`.
  - Efficient **merging and joining** using `merge()` and `join()`.

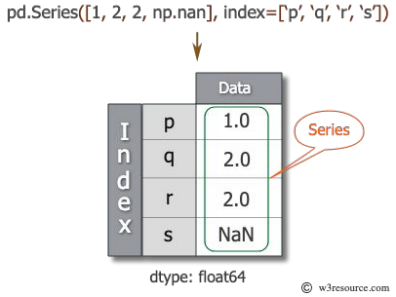




Comparison with Other Tools

- **Pandas vs Excel**
  - More powerful for **automated, repeatable workflows**.
  - Handles **larger datasets** more efficiently.
- **Pandas vs SQL**
  - SQL for **querying databases**; Pandas for in-memory, **flexible data analysis**.
  - **Both can complement** each other using Pandas' `pd.read_sql()`.
- **Pandas vs Other Python Libraries (e.g., NumPy)**
  - **Pandas** built on **NumPy**, adds easier handling for tabular data.
  - Higher-level API than **NumPy** for **data wrangling tasks**.





Install Pandas

SPECIALIZATION

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
PANDAS AND NUMPY

LECTURE 1

PANDAS

Installing Pandas

- **Using pip**

```bash
pip install pandas
```

  - Most common method for standard Python installations.

- **Using Anaconda Distribution**

```bash
conda install pandas
```

  - Recommended for **data science environments** as it includes Pandas and other libraries (NumPy, SciPy).

Setting Up the Environment

- **Python Version Compatibility**
  - Pandas works with **Python 3.7 and above**.
  - Verify Python version:

```bash
python --version
```

- **Importing Pandas**
  - Import Pandas conventionally:

```python
import pandas as pd
```

  - `pd` is a commonly used alias for easier access to Pandas functions.

Checking the Installation

- **Verify Pandas Installation**
  - Check if Pandas is installed successfully by checking the version:

```python
import pandas as pd
print(pd.__version__)
```

  - Ensures that the latest or correct version is installed.

Pandas Data Structures Series

SPECIALIZATION

CS316
INTRODUCTION TO AI AND DATA SCIENCE

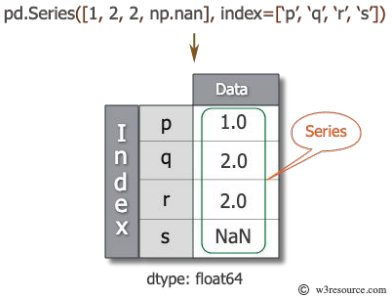
CHAPTER 3
PANDAS AND NUMPY

LECTURE 1

PANDAS

Pandas Series – Definition and Characteristics

- What is a Series?
  - **One-dimensional** labeled array, capable of holding any data type.
  - Similar to a **column in a DataFrame** or **an array** in NumPy.
- Characteristics
  - Data is aligned to an **index** (row labels).
  - **Heterogeneous data** types: integers, floats, strings, etc.
  - Supports **vectorized operations** (element-wise).



Creating a Series

- From a List

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

  - Automatically assigns **default integer index** (0, 1, 2, 3).

- From a Dictionary

```python
data = {'a': 10, 'b': 20, 'c': 30}
s = pd.Series(data)
```

  - Index is derived from **dictionary keys**.

Creating a Series

From a List with Custom Index: You can specify a custom index when creating a Series from a list.

python Copy code

```python
import pandas as pd
s = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
```

From a NumPy Array: You can also create a `Series` from a NumPy array.

python Copy code

```python
import numpy as np
import pandas as pd
data = np.array([1, 2, 3, 4])
s = pd.Series(data)
```

Creating a Series

3. **From a Scalar Value:** A `Series` can also be created from a scalar value, which will be repeated for each index.

```python
import pandas as pd
s = pd.Series(5, index=[0, 1, 2, 3])
```

4. **From Another Series:** You can copy or create a new `Series` from an existing one.

```python
import pandas as pd
original_series = pd.Series([10, 20, 30], index=['x', 'y', 'z'])
new_series = pd.Series(original_series)
```

Accessing Data in a Series

- **Access by Index Label**

```python
s['a']
```

  - Retrieves value associated with index `'a'`.

- **Access by Position**

```python
s[0]
```

  - Retrieves the **first element** in the Series.

- **Slicing**

```python
s[1:3]
```

  - Retrieves a subset of the Series (positions 1 and 2).

Slicing in Pandas Series

- **What is Slicing?**
  - Extracting a subset of elements from a Series based on **index positions** or **labels**.
  - Allows accessing **multiple elements** at once.

- **Slicing by Position (Integer-based indexing)**

```python
s[1:4]
```

  - Returns elements from **index 1 to 3** (excludes end index 4).
  - **0-based indexing**.

Slicing in Pandas Series

- **What is Slicing?**
  - Extracting a subset of elements from a Series based on **index positions** or **labels**.
  - Allows accessing **multiple elements** at once.

- **Slicing by Position (Integer-based indexing)**

```python
s[1:4]
```

  - Returns elements from **index 1 to 3** (excludes end index 4).
  - **0-based indexing**.

Slicing in Pandas Series

- **Slicing by Label (Label-based indexing)**

```python
s['a':'c']
```

  - Returns elements from **'a' to 'c'** (inclusive).

- **Slicing with Steps**

```python
s[0:5:2]
```

  - Retrieves every **2nd element** from index 0 to 4.

Pandas Data Structures
Data Frames

SPECIALIZATION

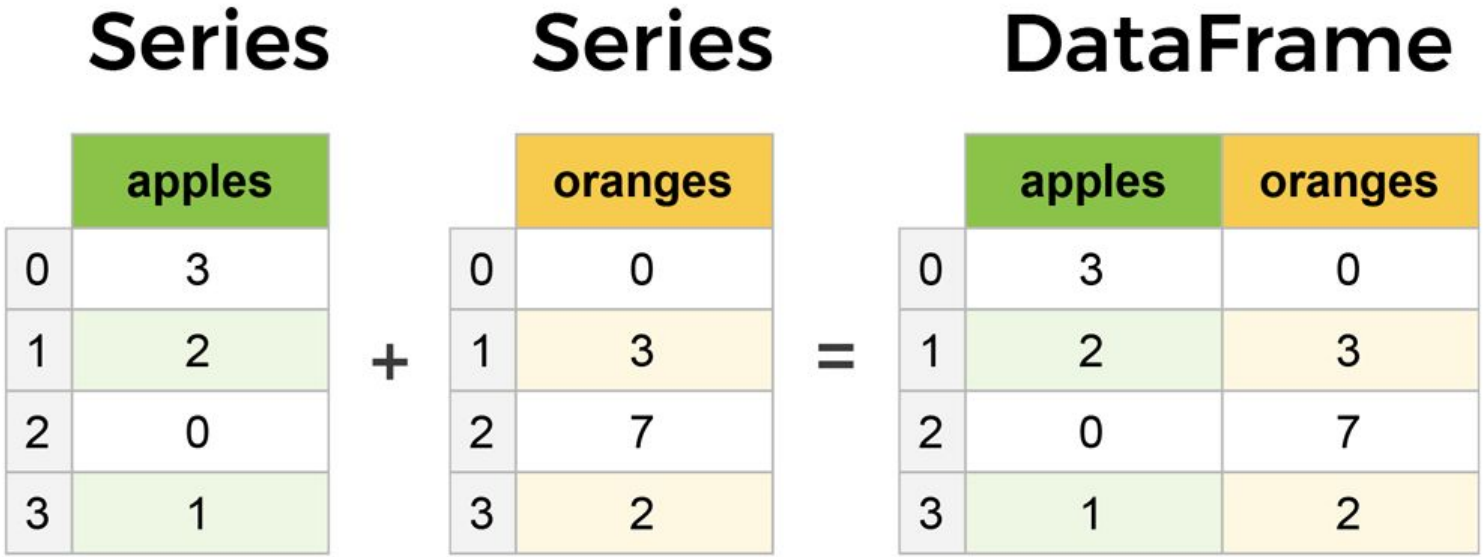
CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
PANDAS AND NUMPY

LECTURE 1

PANDAS

DataFrame - Definition and Characteristics



DataFrame - Definition and Characteristics

- What is a DataFrame?
  - A **two-dimensional, labeled** data structure in Pandas.
  - **Rows and columns**, similar to a table in a database or an Excel sheet.
  - **Heterogeneous data** types supported across different columns.
- Key Characteristics
  - **Size mutable**: Can add/delete rows or columns.
  - Labeled with **index (rows)** and **columns (headers)**.
  - **Flexible operations** like filtering, grouping, reshaping, and joining.

Creating a DataFrame

- **From a Dictionary**

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

  - Dictionary keys become **column labels**, values are **column data**.

- **From a NumPy Array**

```python
import numpy as np
arr = np.array([[1, 2], [3, 4]])
df = pd.DataFrame(arr, columns=['Col1', 'Col2'])
```

  - Requires assigning **column names** explicitly.

- **From CSV or Excel Files**

```python
df = pd.read_csv('file.csv')
df = pd.read_excel('file.xlsx')
```

  - Pandas provides easy methods for importing **external datasets**.

Accessing Data in a DataFrame

- **Accessing Columns**
  - By column label:

```python
df['Name']
```

  - Multiple columns (returns a DataFrame):

```python
df[['Name', 'Age']]
```

- **Accessing Rows**
  - By position (`iloc[]`):

```python
df.iloc[0]  # First row
```

  - By label (`loc[]`):

```python
df.loc['row_label']
```

Pandas Data Structures

Data Frames

Indexing and Selection

SPECIALIZATION

CS316

INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3

PANDAS AND NUMPY

LECTURE 1

PANDAS

Label-based Indexing

- **Definition**:
  - `loc[]` selects data by **label** or **conditional statements**.
- **Accessing Rows by Label**:

```python
df.loc['row_label']
```

  - Retrieves a row using a **specific label** from the index.
- **Selecting Specific Columns by Label**:

```python
df.loc[:, 'column_label']
```

  - Retrieves an entire **column** by label.
- **Conditional Selection**:

```python
df.loc[df['Age'] > 25]
```

  - Retrieves rows where the condition is **True**.

Label-based Indexing

Subsetting with `.loc[]`

- Subset specific rows and columns using `.loc[]` (label-based selection).

python Copy code

```python
df.loc[:, ['column1', 'column2']]
```

- Example:

python Copy code

```python
df.loc[:, ['Name', 'Salary']]  # Select all rows for 'Name' and 'Salary'
```

Label-based Indexing

Filtering by Multiple Conditions

- Combine conditions using logical operators:
  - `&` for AND
  - `|` for OR

```python
df[(df['Age'] > 30) & (df['Gender'] == 'Male')]
```

- Example:

```python
df[(df['Age'] > 30) & (df['Salary'] > 50000)]  # Rows where 'Age' > 30 AND 'Sa
```

Integer-based Indexing

- **Definition**:
  - `iloc[]` selects data based on **integer position** (like NumPy arrays).
- **Accessing Rows by Position**:

```python
df.iloc[0]
```

  - Retrieves the **first row** (0-based index).
- **Selecting a Range of Rows and Columns**:

```python
df.iloc[0:3, 1:3]
```

  - Selects rows **0 to 2** and columns **1 to 2**.

Pandas Data Structures
Data Importing in Pandas

SPECIALIZATION

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
PANDAS AND NUMPY

LECTURE 1

PANDAS

Reading Data into Pandas - From CSV

1. **From CSV Files (`pd.read_csv()`)**
   - **Description:**
     - CSV (Comma-Separated Values) is one of the most common data formats.
     - `pd.read_csv()` is highly flexible, capable of handling large datasets efficiently.
   - **Basic Usage:**

```python
df = pd.read_csv('data.csv')
```

Reading Data into Pandas - From CSV

- `sep` : Specifies delimiter (default is a comma). For tab-separated values:

```python
df = pd.read_csv('data.tsv', sep='\t')
```

- `header` : Defines which row is treated as the header:

```python
df = pd.read_csv('data.csv', header=0)  # First row is the header
```

- `index_col` : Set a specific column as the DataFrame index:

```python
df = pd.read_csv('data.csv', index_col='ID')
```

- `usecols` : Read only specific columns:

```python
df = pd.read_csv('data.csv', usecols=['Name', 'Age'])
```

Common Parameters

Reading Data into Pandas - From Excel Files

- **Description**:
  - Excel is widely used for data storage and analysis. Pandas allows reading `.xlsx` and `.xls` formats.
  - Requires the **openpyxl** or **xlrd** library for handling Excel files.
- **Basic Usage**:

python Copy code

```
df = pd.read_excel('data.xlsx')
```

Reading Data into Pandas - From Excel Files

- `sheet_name` : Specifies the sheet to read (can be sheet name or index):

```python
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')
```

  - Use `sheet_name=None` to read **all sheets** into a dictionary.

- `skiprows` : Skip specific rows at the beginning of the file:

```python
df = pd.read_excel('data.xlsx', skiprows=2)  # Skip the first two rows
```

- `usecols` : Select specific columns to load:

```python
df = pd.read_excel('data.xlsx', usecols='A:C')  # Load columns A to C
```

Common Parameters

Reading Data into Pandas – From SQL Databases

- **Description**:
  - SQL databases store structured data, and Pandas can directly query relational databases such as SQLite, MySQL, PostgreSQL, and more.
  - Pandas requires a **database connection** to execute SQL queries.
- **Basic Usage (SQLite Example)**:

python Copy code

```
import sqlite3
conn = sqlite3.connect('database.db')
df = pd.read_sql('SELECT * FROM table_name', conn)
```

Reading Data into Pandas - From SQL Databases

- For PostgreSQL/MySQL:
  - Use libraries like `psycopg2` (PostgreSQL) or `mysql-connector-python` (MySQL):

```python
import psycopg2
conn = psycopg2.connect("dbname=test user=postgres password=secret")
df = pd.read_sql('SELECT * FROM table_name', conn)
```

- Parameter `params`:
  - Use with parameterized SQL queries:

```python
df = pd.read_sql('SELECT * FROM table WHERE Age > ?', conn, params=[30]
```

Reading Data into Pandas - From JSON Files

- **Description**:
  - JSON (JavaScript Object Notation) is a lightweight format often used for web APIs.
  - Pandas can read JSON into a DataFrame, converting nested structures into rows and columns.
- **Basic Usage**:

```python
df = pd.read_json('data.json')
```

Reading Data into Pandas – From JSON Files

- `orient` : Specifies the expected structure of the JSON file. Common options:
  - `records` : List of dictionaries (default format).
  - `index` : Data formatted as a dictionary of dictionaries, with row labels.

```python
df = pd.read_json('data.json', orient='index')
```

- `lines` : Set to `True` if JSON is in newline-delimited format:

```python
df = pd.read_json('data.json', lines=True)
```

Reading Data into Pandas - From JSON Files

`orient` Parameter

The `orient` parameter specifies the expected structure (or orientation) of the JSON file. The most common options include:

1. `records` **(default option)**:
   - The JSON file is expected to be a **list of dictionaries**, where each dictionary represents a row in the `DataFrame`.
   - **Structure Example**:

```json
[
    {"column1": value1, "column2": value2},
    {"column1": value3, "column2": value4}
]
```

   - **Code Example**:

```python
df = pd.read_json('data.json', orient='records')
```

Reading Data into Pandas - From JSON Files

2. `index`:

   - The JSON file is expected to be a **dictionary of dictionaries**, where the keys of the outer dictionary represent the row labels (index), and the inner dictionaries contain the column data for each row.
   - **Structure Example**:

```json
{
    "row1": {"column1": value1, "column2": value2},
    "row2": {"column1": value3, "column2": value4}
}
```

   - **Code Example**:

```python
df = pd.read_json('data.json', orient='index')
```

Reading Data into Pandas - From JSON Files

`lines` Parameter

The `lines=True` parameter is used when the JSON file is in a **newline-delimited format**. In this format, each line in the file represents a separate JSON object.

- **Structure Example**:

```json
{"column1": value1, "column2": value2}
{"column1": value3, "column2": value4}
```

- **Code Example**:

```python
df = pd.read_json('data.json', lines=True)
```

Pandas Data Structures
Data Exporting in Pandas

SPECIALIZATION

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
PANDAS AND NUMPY

LECTURE 1

PANDAS

Data Exporting in Pandas – Exporting to CSV

1. Exporting to CSV

- **Basic Usage:**
  - Save a DataFrame to a CSV file.

```python
df.to_csv('output.csv', index=False)
```

Data Exporting in Pandas - Exporting to CSV

- `sep` : Specify a different delimiter (e.g., tab-separated values `\t`).

```python
df.to_csv('output.tsv', sep='\t', index=False)
```

- `header` : Whether to include the column headers in the output file.

```python
df.to_csv('output.csv', header=False)  # Excludes column names
```

- `index` : Include or exclude the row index in the file.

```python
df.to_csv('output.csv', index=True)  # Includes row indices
```

- `columns` : Select specific columns to save in the CSV.

```python
df.to_csv('output.csv', columns=['Name', 'Age'], index=False)
```

Data Exporting in Pandas - Exporting to CSV

Advanced Parameters:

- `line_terminator` : Customize the line ending (e.g., `\n` , `\r\n`).

```python
df.to_csv('output.csv', line_terminator='\n')
```

- `encoding` : Specify character encoding (useful for non-ASCII data).

```python
df.to_csv('output.csv', encoding='utf-8')
```

Data Exporting in Pandas – Exporting to Excel

2. Exporting to Excel

- Basic Usage:
  - Save a DataFrame to an Excel file.

python Copy code

```python
df.to_excel('output.xlsx', sheet_name='Sheet1', index=False)
```

Data Exporting in Pandas – Exporting to Excel

- `sheet_name` : Set the name of the Excel sheet where data will be written.

```python
df.to_excel('output.xlsx', sheet_name='Data', index=False)
```

- `index` : Include or exclude the row index in the Excel file.

```python
df.to_excel('output.xlsx', index=True)  # Includes row indices
```

Data Exporting in Pandas – Exporting to Excel

- **Advanced Parameters**:
  - `engine`: Specify the engine for writing Excel files (usually **openpyxl** for `.xlsx`).

```python
df.to_excel('output.xlsx', engine='openpyxl')
```

  - `startrow` **and** `startcol`: Define the starting row and column for writing data.

```python
df.to_excel('output.xlsx', startrow=2, startcol=1)
```

    - This is useful when you want to write data to a specific location in the sheet.
  - `freeze_panes`: Freeze the top or side rows/columns.

```python
df.to_excel('output.xlsx', freeze_panes=(1, 0))  # Freeze the top row
```

  - `columns`: Select specific columns to save.

```python
df.to_excel('output.xlsx', columns=['Name', 'Salary'])
```

Data Exporting in Pandas – Exporting to SQL

- **Basic Usage:**
  - Save a DataFrame to an SQL table.

```python
df.to_sql('table_name', conn, if_exists='replace', index=False)
```

Data Exporting in Pandas – Exporting to SQL

- `conn` : Database connection object (e.g., from SQLite, MySQL, PostgreSQL).

```python
import sqlite3
conn = sqlite3.connect('database.db')
df.to_sql('employees', conn, index=False)
```

- `if_exists` : What to do if the table already exists.
  - `'replace'` : Drop the table, then recreate and insert data.
  - `'append'` : Add the DataFrame to the existing table.
  - `'fail'` : Raise an error if the table exists.

```python
df.to_sql('employees', conn, if_exists='append', index=False)
```

Data Exporting in Pandas – Exporting to SQL

Advanced Parameters:

- `dtype`: Specify the SQL data type for each column.

```python
df.to_sql('employees', conn, dtype={'Name': String, 'Age': Integer})
```

  - Useful when you want to control how columns are stored in the database.

- `chunksize`: Write the DataFrame in batches to handle large datasets.

```python
df.to_sql('employees', conn, chunksize=500)
```

  - Writes 500 rows at a time, useful for exporting large DataFrames.

- `method`: Specify the SQL insert method.

  - You can optimize bulk inserts using specific database features.

```python
df.to_sql('employees', conn, method='multi')
```

Data Exporting in Pandas – Exporting to JSON

Basic Usage

- Save a DataFrame to a JSON file.

python Copy code

```python
df.to_json('output.json')
```

Data Exporting in Pandas – Exporting to JSON

- `orient` : Controls the structure (format) of the output JSON. Key formats:
  - `'columns'` (default): Columns as keys, with nested row values.

```python
df.to_json('output.json', orient='columns')
```

Output:

```json
{
  "column1": {"0": "value1", "1": "value2"},
  "column2": {"0": "value3", "1": "value4"}
}
```

Data Exporting in Pandas – Exporting to JSON

- `'records'`: List of dictionaries (each row is a dictionary).

```python
df.to_json('output.json', orient='records')
```

Output:

```json
[
  {"column1": "value1", "column2": "value3"},
  {"column1": "value2", "column2": "value4"}
]
```

Data Exporting in Pandas – Exporting to JSON

- `'index'`: Row indices are the keys, with rows as nested dictionaries.

```python
df.to_json('output.json', orient='index')
```

Output:

```json
{
  "0": {"column1": "value1", "column2": "value3"},
  "1": {"column1": "value2", "column2": "value4"}
}
```

Data Exporting in Pandas – Exporting to JSON

- `lines` : Writes the JSON data in a **newline-delimited** format (useful for large datasets and streaming).

```python
df.to_json('output.json', lines=True, orient='records')
```

Output:

```json
{"column1": "value1", "column2": "value3"}
{"column1": "value2", "column2": "value4"}
```

- Each row is written as a separate JSON object, making it easier to process with streaming applications or when reading line-by-line.

Data Exporting in Pandas – Exporting to JSON

- `date_format` : Control the format of datetime objects in the JSON output.
  - Use `'iso'` for ISO 8601 format or `'epoch'` for UNIX timestamps.

```python
df.to_json('output.json', date_format='iso')
```

  - Outputs dates as `"YYYY-MM-DDTHH:MM:SS"`.
- `double_precision` : Control the precision of floating-point numbers.

```python
df.to_json('output.json', double_precision=2)
```

  - Rounds floating-point numbers to 2 decimal places.
- `force_ascii` : Force encoding to ASCII (useful for non-ASCII characters like emojis).

```python
df.to_json('output.json', force_ascii=False)
```

  - Allows Unicode characters to appear as-is instead of ASCII-encoded characters.

Common Use Cases

1. **Exporting Data to Share**:
    - **CSV**: Best for sharing raw data in plain text format (widely used across applications).
    - **Excel**: Best for reports where formatting, multiple sheets, and formulas may be needed.
    - **SQL**: Best for saving structured data into a relational database for further querying.
2. **Performance Tips**:
    - For **large datasets**, export in **chunks** using `chunksize` to avoid memory issues.
    - When exporting to **Excel**, consider limiting the file size and using CSV for very large datasets.

Pandas Data Structures
Handling File Paths and URLs

SPECIALIZATION

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
PANDAS AND NUMPY

LECTURE 1

PANDAS

Handling File Paths

- **Local File Paths**:
  - When working with local files, simply provide the **relative** or **absolute path** to the file.

```python
df = pd.read_csv('data/myfile.csv')  # Relative path
df = pd.read_csv('/home/user/data/myfile.csv')  # Absolute path
```

- **Working with Different Operating Systems**:
  - Use Python's `os` library for **cross-platform compatibility**.

```python
import os
path = os.path.join('data', 'myfile.csv')  # Automatically handles slashes
df = pd.read_csv(path)
```

- **Using `pathlib` for File Handling**:
  - `pathlib` is a modern and intuitive way to handle paths.

```python
from pathlib import Path
path = Path('data/myfile.csv')
df = pd.read_csv(path)
```

Handling URLs

- **Reading Data from a URL**:
  - Pandas can directly read files hosted on the web, such as CSV or Excel files, using a URL.

```python
df = pd.read_csv('https://example.com/data.csv')
```

- **Supported File Types for URLs**:
  - **CSV**: `pd.read_csv()`
  - **Excel**: `pd.read_excel()`
  - **JSON**: `pd.read_json()`
  - Example:

```python
df = pd.read_json('https://example.com/data.json')
```

Handling URLs

- **Handling Authentication for URLs:**
  - For URLs that require authentication (e.g., APIs), use `requests` to handle the download and pass it to Pandas.

python Copy code

```
import requests
response = requests.get('https://example.com/data.csv', auth=('username', 'pass
with open('localfile.csv', 'wb') as f:
    f.write(response.content)
df = pd.read_csv('localfile.csv')
```

Advanced URL Handling

- **Reading Data from APIs**:
  - Combine Pandas with **web APIs** to read data directly from the web.

python Copy code

```python
import requests
response = requests.get('https://api.example.com/data')
data = response.json()
df = pd.DataFrame(data)
```

- **Reading from Cloud Storage**:
  - Pandas supports reading from cloud storage systems (AWS S3, Google Cloud Storage, etc.) using specialized libraries (e.g., **s3fs** for AWS).

python Copy code

```python
df = pd.read_csv('s3://bucket_name/data.csv', storage_options={'key': 'your_key
```

جامعة الامير سلطان
PRINCE SULTAN UNIVERSITY

FOR

SPECIALIZATION

Robotics & Internet-of-Things

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
Pandas and Numpy Data Structures

LECTURE 2
Data Manipulation in Pandas

Prof. Anis Koubaa

SEP 2024

Importance of Data Manipulation in Data Science

- **Data Cleaning**: Prepares messy, raw data for analysis by handling missing values, duplicates, and inconsistencies.
- **Feature Selection**: Helps focus on key variables and extract relevant information for better insights.
- **Data Transformation**: Reshapes and scales data for advanced analysis and machine learning models.
- **Improved Decision Making**: Enables more accurate, data-driven insights, leading to better decisions.

Selecting and Filtering Data

Selecting and Filtering Data- Subsetting Columns

- **Select a Single Column**:
  - To select the `name` column:

```python
df['name']
```

Example: Extract the names of all students.

```python
0      Mohamed Salah
1                NaN
2       Ahmed Assiri
3    Abdullah Ahmari
4        Faisal Ali
```

- **Select Multiple Columns**:
  - To select `student_id`, `name`, and `final` columns:

```python
df[['student_id', 'name', 'final']]
```

	student_id	name	major01	major02	project	final
0	120060258	Mohamed Salah	0.25	0.00	0.000	13.0
1	211110888	NaN	15.22	10.75	14.670	27.0
2	211210095	Ahmed Assiri	13.25	NaN	16.250	NaN
3	212110756	Abdullah Ahmari	5.91	1.25	5.000	7.0
4	212110889	Faisal Ali	16.00	NaN	6.670	NaN
5	212110977	NaN	10.31	12.45	15.758	33.0
6	213110290	Abdulaziz Samir	15.00	NaN	NaN	NaN
7	213110419	Anas Karim	16.00	12.63	16.670	23.0
8	213110460	Mahmoud Adel	10.00	6.38	NaN	8.0
9	213110964	NaN	10.00	7.50	11.670	15.0
10	213110964	Hany Abdullah	9.91	NaN	5.330	30.0
11	213111074	Abdelhalim Mohanned	10.56	12.00	11.330	20.0
12	213111128	NaN	19.00	17.25	18.000	NaN
13	213210082	Khaled Majid	20.00	18.88	18.670	37.0
14	213254033	Saad Ali	10.00	NaN	NaN	3.0

Example:

student_id	name	final
120060258	Mohamed Salah	13.0
211110888	NaN	27.0
211210095	Ahmed Assiri	NaN
212110756	Abdullah Ahmari	7.0

Filtering Rows Based on Conditions

- **Filter Based on a Single Condition**:
  - Select rows where `major01` is greater than 15.

```python
df[df['major01'] > 15]
```

Example: Get students whose `major01` score exceeds 15.

student_id	name	major01	major02	project	final
212110889	Faisal Ali	16.00	NaN	6.670	NaN
213111128	NaN	19.00	17.25	18.000	NaN
213210082	Khaled Majid	20.00	18.88	18.670	37.0

- **Filter with Multiple Conditions**:
  - Find students who scored more than 10 in both `major01` and `project`:

```python
df[(df['major01'] > 10) & (df['project'] > 10)]
```

Example:

student_id	name	major01	major02	project	final
211110888	NaN	15.22	10.75	14.670	27.0
211210095	Ahmed Assiri	13.25	NaN	16.250	NaN
213210082	Khaled Majid	20.00	18.88	18.670	37.0

Filtering Rows Based on Conditions

- **Filtering for Missing Data (`NaN`):**
  - Select rows where `final` is missing:

```python
df[df['final'].isnull()]
```

Copy code

Example:

student_id	name	major01	major02	project	final
211210095	Ahmed Assiri	13.25	NaN	16.250	NaN
212110889	Faisal Ali	16.00	NaN	6.670	NaN
213111128	NaN	19.00	17.25	18.000	NaN

Filtering Rows Based on Multiple Conditions

- **Multiple Conditions:**
  - Filter rows where `major01` is greater than 10 and `project` is greater than 10:

```python
df[(df['major01'] > 10) & (df['project'] > 10)]
```

Copy code

Example:

student_id	name	major01	project
211110888	NaN	15.22	14.670
211210095	Ahmed Assiri	13.25	16.250
213210082	Khaled Majid	20.00	18.670

Combining Subsetting and Filtering

- **Combining Subsetting and Filtering:**
  - Select specific columns (`student_id` , `name`) for students where `final` score is greater than 25:

```python
df.loc[df['final'] > 25, ['student_id', 'name']]
```

Copy code

Example:

student_id	name
211110888	NaN
213210082	Khaled Majid

Selecting Specific Rows and Columns by Position

- **Selecting Specific Rows and Columns by Position**:
  - Using `.iloc[]` to select **rows by position** and **columns by position**:

```python
df.iloc[0:3, [0, 1, 5]]  # First 3 rows, select columns at index 0, 1, and 5
```

Example:

student_id	name	final
120060258	Mohamed Salah	13.0
211110888	NaN	27.0
211210095	Ahmed Assiri	NaN

- **Selecting a Single Row by Position**:
  - Use `.iloc[]` to get a single row by index position (e.g., 2nd row):

```python
df.iloc[1]
```

Using .query() for Readable Filtering:

- **Using `.query()` for Readable Filtering:**
  - `.query()` uses SQL-like syntax to filter data, making conditions more readable.
  - Example: Select students where `major01` is greater than 10 and `final` score is greater than 20.

```python
df.query('major01 > 10 and final > 20')
```

Copy code

Output:

student_id	name	major01	major02	project	final
211110888	NaN	15.22	10.75	14.670	27.0

Handling Missing Data

Handling Missing Data in Python

Why Handle Missing Data?

- Missing data can distort analysis and lead to biased results.
- Proper handling ensures data integrity and better model performance.

	student_id	name	major01	major02	project	final
0	120060258	Mohamed Salah	0.25	0.00	0.000	13.0
1	211110888	NaN	15.22	10.75	14.670	27.0
2	211210095	Ahmed Assiri	13.25	NaN	16.250	NaN
3	212110756	Abdullah Ahmari	5.91	1.25	5.000	7.0
4	212110889	Faisal Ali	16.00	NaN	6.670	NaN
5	212110977	NaN	10.31	12.45	15.758	33.0
6	213110290	Abdulaziz Samir	15.00	NaN	NaN	NaN
7	213110419	Anas Karim	16.00	12.63	16.670	23.0
8	213110460	Mahmoud Adel	10.00	6.38	NaN	8.0
9	213110964	NaN	10.00	7.50	11.670	15.0
10	213110964	Hany Abdullah	9.91	NaN	5.330	30.0
11	213111074	Abdelhalim Mohanned	10.56	12.00	11.330	20.0
12	213111128	NaN	19.00	17.25	18.000	NaN
13	213210082	Khaled Majid	20.00	18.88	18.670	37.0
14	213254033	Saad Ali	10.00	NaN	NaN	3.0

Key Alternatives for Handling Missing Data

1. **Drop Missing Data**
   - **When**: If missing values are few or non-critical.
   - **Impact**: Reduces dataset size but avoids introducing bias.
2. **Fill Missing Data (Imputation)**
   - **Options**: Replace missing values with the mean, median, or a constant.
   - **Impact**: Retains all data, but assumptions must be valid to avoid bias.
3. **Forward/Backward Fill**
   - **When**: Use prior or subsequent data points to fill gaps.
   - **Impact**: Best for time series or sequential data.

Handling Missing Data in Python

+ Code + Text

[2]
```
# Search for an email address in the text using regex
email = find_email_regex(text)
print(f"Found email: {email}")

```

```
Found email: support@example.com
```

[ ]
```
from google.colab import drive
drive.mount('/content/drive')
```

```
dataset_path='/content/drive/My Drive/CS313: Data Science Course/Topic 2: Python Data Structures and Viz/pandas/dataset/'
file_name = 'grade_book_missing_data.csv'
missing_grades_path=dataset_path+file_name
print(missing_grades_path)

grades_data_frame = pd.read_csv(missing_grades_path)
grades_data_frame
```

```
/content/drive/My Drive/CS313: Data Science Course/Topic 2: Python Data Structures and Viz/pandas/dataset/grade_book_missing_data.csv
```

	student_id	name	major01	major02	project	final
0	120060258	Mohamed Salah	0.25	0.00	0.000	13.0
1	211110888	NaN	15.22	10.75	14.670	27.0
2	211210095	Ahmed Assiri	13.25	NaN	16.250	NaN
3	212110756	Abdullah Ahmari	5.91	1.25	5.000	7.0
4	212110889	Faisal Ali	16.00	NaN	6.670	NaN
5	212110977	NaN	10.31	12.45	15.758	33.0
6	213110290	Abdulaziz Samir	15.00	NaN	NaN	NaN

Identifying Missing Values

- **Checking for Missing Data**:
  - Use `.isnull()` to identify missing values in the DataFrame:

```python
df.isnull()
```

Copy code

Output (partial):

student_id	name	major01	major02	project	final
120060258	False	False	False	False	False
211110888	True	False	False	False	False
211210095	False	False	True	False	True

Summary of Missing Values

- Use `.isnull().sum()` to get the total count of missing values per column:

```python
df.isnull().sum()
```

Copy code

Output:

Column	Missing Values
name	5
major01	0
major02	5
project	3
final	4

Filling Missing Values (fillna())

- **Filling with a Constant Value**:
  - Replace all missing values in the `final` column with `0`:

```python
df['final'].fillna(0, inplace=True)
```

Example Before:

name	final
Ahmed Assiri	NaN
Khaled Majid	37.0

Example After:

name	final
Ahmed Assiri	0.0
Khaled Majid	37.0

Filling Missing Values (fillna())

Filling with the Mean or Median:

- Fill missing values in `major02` with the column's **mean**:

```python
df['major02'].fillna(df['major02'].mean(), inplace=True)
```

Example Before:

name	major02
Ahmed Assiri	NaN
Mahmoud Adel	6.38

Example After:

name	major02
Ahmed Assiri	11.73
Mahmoud Adel	6.38

Dropping Missing Values (dropna())

- **Drop Rows with Missing Values**:
  - Remove rows where **any** column has a missing value:

```python
df.dropna(inplace=True)
```

Example Before:

name	final
Faisal Ali	NaN
Hany Abdullah	30.0

Example After (Faisal Ali's row is dropped):

name	final
Hany Abdullah	30.0

- **Drop Rows with Missing Values in Specific Columns**:
  - Drop rows where the `final` score is missing:

```python
df.dropna(subset=['final'], inplace=True)
```

Dropping Missing Values (dropna())

- **Remove Rows/Columns with Missing Data:**
  - Use `dropna()` to remove rows or columns with missing values.

```python
df.dropna(axis=0, inplace=True)  # Drops rows
df.dropna(axis=1, inplace=True)  # Drops columns
```

  - Customize with `thresh=` to keep rows/columns with a minimum number of non-null values.

Dropping Missing Values (dropna())

```python
import pandas as pd

# Sample DataFrame
df = pd.DataFrame({
    'A': [1, 2, None, 4],
    'B': [None, 2, 3, 4],
    'C': [1, None, 3, 4]
})

print('df\n',df,'\n')

# Detect missing values
missing_values = df.isnull()
# Removing rows with missing data
df_dropna = df.dropna()
print('df_dropna\n', df_dropna,'\n')

# Filling missing data with a value
df_filled = df.fillna(value=0)
print('df_filled\n',df_filled,'\n')

# Filling with the column mean
df_fillmean = df.fillna(df.mean())
print('df_fillmean\n',df_fillmean,'\n')

```

```
df
     A    B    C
0  1.0  NaN  1.0
1  2.0  2.0  NaN
2  NaN  3.0  3.0
3  4.0  4.0  4.0

df_dropna
     A    B    C
3  4.0  4.0  4.0

df_filled
     A    B    C
0  1.0  0.0  1.0
1  2.0  2.0  0.0
2  0.0  3.0  3.0
3  4.0  4.0  4.0

df_fillmean
          A    B         C
0  1.000000  3.0  1.000000
1  2.000000  2.0  2.666667
2  2.333333  3.0  3.000000
3  4.000000  4.0  4.000000
```

Data Transformation

Summary of Data Transformation

- **Add/Remove**: Easily add or delete columns using column operations.
- **Rename**: Use `rename()` for renaming columns.
- **Change Data Types**: Use `astype()` to convert data types.
- **Apply Functions**: Use `apply()` for applying functions to columns and `map()` for element-wise transformations.

Adding Columns

- **Adding a New Column**:
  - Example: Create a new column `total_score` by summing `project` and `final`:

```python
df['total_score'] = df['project'] + df['final']
```

Before:

student_id	name	project	final
120060258	Mohamed Salah	0.000	13.0
211110888	NaN	14.670	27.0

After:

student_id	name	project	final	total_score
120060258	Mohamed Salah	0.000	13.0	13.0
211110888	NaN	14.670	27.0	41.67

Removing Columns

Removing a Column:

- Remove the `major02` column:

```python
df.drop(columns=['major02'], inplace=True)
```

Renaming Columns (rename())

- **Renaming a Single Column**:
  - Rename `student_id` to `ID`:

```python
df.rename(columns={'student_id': 'ID'}, inplace=True)
```

Before:

student_id	name	final
120060258	Mohamed Salah	13.0

After:

ID	name	final
120060258	Mohamed Salah	13.0

- **Renaming Multiple Columns**:
  - Rename `major01` to `Major1` and `final` to `Final_Score`:

```python
df.rename(columns={'major01': 'Major1', 'final': 'Final_Score'}, inplace=True)
```

Changing Data Types (astype())

- **Changing Column Data Types**:
  - Convert `student_id` to a string data type:

```python
df['student_id'] = df['student_id'].astype(str)
```

- **Handling Missing Data When Changing Types**:
  - Convert `final` column to integer, filling missing values with `0`:

```python
df['final'] = df['final'].fillna(0).astype(int)
```

Before:

name	final
Ahmed Assiri	NaN
Khaled Majid	37.0

After:

name	final
Ahmed Assiri	0
Khaled Majid	37

Applying Functions (apply(), map())

- **Using `apply()` for Column-Wise Operations:**
  - Apply a function to calculate the percentage of `final` scores (out of 40):

```python
df['final_percentage'] = df['final'].apply(lambda x: (x / 40) * 100)
```

Before:

name	final
Mohamed Salah	13.0

After:

name	final	final_percentage
Mohamed Salah	13.0	32.5

Applying Functions (apply(), map())

Using `map()` for Element-wise Mapping:

- Replace `NaN` values in the `name` column with "Unknown":

```python
df['name'] = df['name'].map(lambda x: 'Unknown' if pd.isnull(x) else x)
```

Before:

name	final
NaN	27.0

After:

name	final
Unknown	27.0

Sorting and Ranking Data in Pandas

Summary of Sorting and Ranking

- **Sort Data**: Use `sort_values()` for sorting data by one or more columns, with control over the sort order and placement of missing values.
- **Rank Data**: Use `rank()` to rank data with options to handle ties and rank in ascending or descending order.

Sorting Data (sort_values())

Sorting by a Single Column:

- Sort the data by `final` scores in **ascending order**:

```python
df.sort_values(by='final', ascending=True)
```

Before Sorting:

name	final
Mohamed Salah	13.0
Khaled Majid	37.0

After Sorting:

name	final
Abdullah Ahmari	7.0
Mohamed Salah	13.0
Khaled Majid	37.0

Sorting Data (sort_values())

Sorting by Multiple Columns:

- Sort by `major01` in descending order and then by `final` in ascending order:

```python
df.sort_values(by=['major01', 'final'], ascending=[False, True])
```

Example (sorted by highest `major01`, then lowest `final`):

name	major01	final
Khaled Majid	20.00	37.0
Faisal Ali	16.00	NaN
Abdullah Ahmari	5.91	7.0

Sorting Data (sort_values())

- **Sorting with Missing Values**:
  - By default, missing values (`NaN`) are placed at the **end** when sorting. To place them at the **beginning**:

```python
df.sort_values(by='final', na_position='first')
```

Ranking Data (rank())

Basic Ranking:

- Rank the `final` scores (default is **average ranking**):

```python
df['final_rank'] = df['final'].rank()
```

Before Ranking:

name	final
Mohamed Salah	13.0
Khaled Majid	37.0

After Ranking:

name	final	final_rank
Mohamed Salah	13.0	2.0
Khaled Majid	37.0	5.0

Ranking Data (rank())

Handling Ties in Ranking:

- **Average** ranking (default): Assigns the average rank to tied values.

```python
df['final_rank'] = df['final'].rank(method='average')
```

- **Min** ranking: Assigns the **lowest** rank to tied values.

```python
df['final_rank'] = df['final'].rank(method='min')
```

- **Max** ranking: Assigns the **highest** rank to tied values.

```python
df['final_rank'] = df['final'].rank(method='max')
```

- **Dense** ranking: Similar to `min` but with **no gaps** in rank numbers.

```python
df['final_rank'] = df['final'].rank(method='dense')
```

Ranking Data (rank())

Ranking in Descending Order:

- Rank the data in **descending order** (highest scores get the rank of 1):

```python
df['final_rank'] = df['final'].rank(ascending=False)
```

Copy code

Example:

name	final	final_rank
Khaled Majid	37.0	1.0
Mohamed Salah	13.0	4.0

Group Operations

Group Operations in Pandas

- **GroupBy Mechanics**: Split, apply, and combine data using `groupby()`.
- **Aggregation Functions**: Summarize data using `mean()`, `sum()`, `count()`, and more.
- **Transformation**: Apply transformations to each group and return the same shape.
- **Filtering**: Remove groups based on a condition.

Group Operations in Pandas

	number	student_id	gender	major01	major02	project	final	term	year
0	1	120060257	male	0.25	0.00	0.00	13.0	fall	2015
1	2	211110888	female	15.22	10.75	14.67	27.0	fall	2015
2	3	211210095	male	13.25	15.43	16.25	25.0	fall	2015
3	4	212110756	female	5.91	1.25	5.00	7.0	fall	2015
4	5	212110889	male	11.00	13.00	6.67	27.0	fall	2015

GroupBy Mechanics

- **What is `groupby()`?**
  - `groupby()` is used to group data based on one or more columns and apply operations to each group.
- **Grouping by a Single Column**:
  - Group the data by `gender`:

```python
grouped = df.groupby('gender')
```

  - After grouping, no operation is performed yet; this returns a **GroupBy object** that can be used for further analysis.
- **Grouping by Multiple Columns**:
  - Group by both `gender` and `term`:

```python
grouped = df.groupby(['gender', 'term'])
```

Aggregation Functions

- **Basic Aggregation with `.agg()`:**
  - Calculate the average `final` score for each gender:

```python
df.groupby('gender')['final'].agg('mean')
```

Copy code

Output:

gender	final
female	17.00
male	21.67

Aggregation Functions

Multiple Aggregation Functions:

- Apply multiple aggregation functions (e.g., mean and max) to `major01` for each gender:

```python
df.groupby('gender')['major01'].agg(['mean', 'max'])
```

Copy code

Output:

gender	mean	max
female	10.565	15.22
male	8.83	13.25

Transformation and Filtering

- Using `transform()`:
  - Use `transform()` to apply a function to each group and return a transformed version of the original DataFrame.
  - Example: Subtract the mean of each gender group from the `final` score:

```python
df['final_diff'] = df.groupby('gender')['final'].transform(lambda x: x - x.mean
```

Copy code

Before Transformation:

gender	final
male	13.0
female	27.0

After Transformation (`final_diff` column):

gender	final	final_diff
male	13.0	-8.67
female	27.0	10.0

Filtering Groups with .filter()

- Retain only groups where the average `final` score is greater than 20:

```python
df_filtered = df.groupby('gender').filter(lambda x: x['final'].mean() > 20)
```

Output:

number	student_id	gender	major01	project	final
2	211110888	female	15.22	14.67	27.0
5	212110889	male	11.00	6.67	27.0

Filtering Groups with .filter()

- Retain only groups where the average `final` score is greater than 20:

```python
df_filtered = df.groupby('gender').filter(lambda x: x['final'].mean() > 20)
```

Output:

number	student_id	gender	major01	project	final
2	211110888	female	15.22	14.67	27.0
5	212110889	male	11.00	6.67	27.0

جامعة الامير سلطان
PRINCE SULTAN UNIVERSITY

FOR

SPECIALIZATION

Robotics & Internet-of-Things

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
Pandas and Numpy Data Structures

LECTURE 3
Data Analysis with Pandas

Prof. Anis Koubaa

SEP 2024

Descriptive Statistics for Data Analysis

- **Dataset Overview**: Use `.info()` for metadata on the dataset (non-null values, data types).
- **Summary Statistics**: Use `describe()` for numeric data overview.
- **Mean/Median/Mode**: Calculate individual statistics using `.mean()`, `.median()`, and `.mode()`.
- **Value Counts**: Use `value_counts()` to analyze categorical data.

Descriptive Statistics for Data Analysis

Dataset Overview

number	student_id	gender	major01	major02	project	final	term	year
1	120060257	male	0.25	0.00	0.00	13.0	fall	2015
2	211110888	female	15.22	10.75	14.67	27.0	fall	2015
3	211210095	male	13.25	15.43	16.25	25.0	fall	2015
4	212110756	female	5.91	1.25	5.00	7.0	fall	2015
5	212110889	male	11.00	13.00	6.67	27.0	fall	2015

Descriptive Statistics for Data Analysis

- **Using `.info()` to Get Metadata**:
  - Provides a concise summary of the DataFrame, including data types, non-null counts, and memory usage.

```python
df.info()
```

Output:

```kotlin
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   number      5 non-null      int64
 1   student_id  5 non-null      int64
 2   gender      5 non-null      object
 3   major01     5 non-null      float64
 4   major02     4 non-null      float64
 5   project     5 non-null      float64
 6   final       5 non-null      float64
 7   term        5 non-null      object
dtypes: float64(4), int64(2), object(2)
memory usage: 448.0+ bytes
```

Key Insights from `.info()`:

- Data types of each column (e.g., `int64`, `float64`, `object`).
- Non-null counts, showing how many missing values exist.
- Memory usage of the DataFrame.

Descriptive Statistics for Data Analysis

1. Summary Statistics (`describe()`)

- **Using `describe()` for Summary Statistics:**
  - `describe()` provides a quick summary of the central tendency and spread of the data for numeric columns.

```python
df.describe()
```

Output:

	number	major01	major02	project	final
count	5.0	5.0	4.0	5.0	5.0
mean	3.0	9.93	8.61	8.52	19.8
std	1.58	5.94	7.04	6.32	9.48
min	1.0	0.25	0.00	0.00	7.0
25%	2.0	5.91	3.44	5.00	13.0
50%	3.0	11.00	11.00	6.67	25.0
75%	4.0	13.25	13.38	14.67	27.0
max	5.0	15.22	15.43	16.25	27.0

This gives a comprehensive view of the **count**, **mean**, **standard deviation**, **min**, **max**, and **percentiles** for numeric columns.

Descriptive Statistics for Data Analysis

2. Calculating Mean, Median, Mode

- **Mean**:
  - Calculate the **mean** of the `final` scores:

```python
df['final'].mean()
```

Output:

```
19.8
```

- **Median**:
  - Calculate the **median** of the `major01` scores:

```python
df['major01'].median()
```

Output:

```
11.00
```

Descriptive Statistics for Data Analysis

- **Mode**:
  - Calculate the **mode** of the `final` scores:

```python
df['final'].mode()
```

Output:

```go
0    27.0
dtype: float64
```

Descriptive Statistics for Data Analysis

3. Value Counts

- **Counting Unique Values in a Column**:
  - Use `value_counts()` to count the occurrences of each value in the `gender` column:

```python
df['gender'].value_counts()
```

Output:

gender	count
male	3
female	2

Pandas Data Structures Advanced Statistical Techniques

SPECIALIZATION

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
PANDAS AND NUMPY

LECTURE 1

PANDAS

Summary of Advanced Data Analysis

- **Value Counts**: Use `value_counts()` to count occurrences of categorical values.
- **Correlation and Covariance**: Use `.corr()` and `.cov()` to understand relationships between numeric variables.
- **Cross-Tabulation**: Use `crosstab()` for frequency tables and cross-analysis between categorical variables.

Value Counts

1. Value Counts

- **Counting Unique Values in a Column**:
  - Use `value_counts()` to count the occurrences of each value in the `gender` column:

```python
df['gender'].value_counts()
```

Output:

gender	count
male	3
female	2

- **Normalize the Counts**:
  - Get the **relative frequencies** by setting `normalize=True`:

```python
df['gender'].value_counts(normalize=True)
```

Output:

gender	proportion
male	0.6
female	0.4

Correlation and Covariance

2. Correlation and Covariance

Calculating Correlations

- **Pearson Correlation**:
  - Use `.corr()` to calculate the **Pearson correlation** between numeric columns:

```python
df[['major01', 'major02', 'project', 'final']].corr()
```

Output:

	major01	major02	project	final
major01	1.000	0.843	0.979	0.601
major02	0.843	1.000	0.792	0.759
project	0.979	0.792	1.000	0.692
final	0.601	0.759	0.692	1.000

This matrix shows correlations between `major01`, `major02`, `project`, and `final`. High positive correlations indicate a stronger relationship.

Visualizing with a Heatmap

Visualizing with a Heatmap (Optional, if introducing visualization)

- **Creating a Heatmap for Correlation Matrix:**
  - Use `seaborn` to create a visual representation of the correlation matrix.

python Copy code

```python
import seaborn as sns
import matplotlib.pyplot as plt
sns.heatmap(df[['major01', 'major02', 'project', 'final']].corr(), annot=True,
plt.show()
```

Calculating Covariance

- **Covariance**:
  - Use `.cov()` to compute the **covariance** matrix:

```python
df[['major01', 'major02', 'project', 'final']].cov()
```

Output:

	major01	major02	project	final
major01	35.31	26.33	31.13	16.47
major02	26.33	31.13	22.47	17.47
project	31.13	22.47	32.00	19.00
final	16.47	17.47	19.00	34.67

Calculating Covariance

- **Covariance**:
  - Use `.cov()` to compute the **covariance** matrix:

```python
df[['major01', 'major02', 'project', 'final']].cov()
```

Output:

	major01	major02	project	final
major01	35.31	26.33	31.13	16.47
major02	26.33	31.13	22.47	17.47
project	31.13	22.47	32.00	19.00
final	16.47	17.47	19.00	34.67

Cross Tabulation

Cross-Tabulation

Frequency Tables with `crosstab()`

- **Basic Cross-Tabulation:**
  - Use `pd.crosstab()` to create a **frequency table** for categorical variables, such as `gender` and `term`:

```python
pd.crosstab(df['gender'], df['term'])
```

Copy code

Output:

term	fall
female	2
male	3

Cross-Tabulation

- **Adding Margins to Show Totals:**
  - Use `margins=True` to add row and column totals to the frequency table:

```python
pd.crosstab(df['gender'], df['term'], margins=True)
```

Output:

term	fall	All
female	2	2
male	3	3
All	5	5

Cross-Tabulation

- **Cross-Tabulation with Multiple Variables**:
  - You can also perform cross-tabulation with more than one variable. For instance, count the `final` scores by both `gender` and `term`:

```python
pd.crosstab(df['gender'], df['term'], values=df['final'], aggfunc='mean')
```

Output (average final scores):

term	fall
female	17.0
male	21.67

Cross-Tabulation

- **Cross-Tabulation with Multiple Variables**:
  - You can also perform cross-tabulation with more than one variable. For instance, count the `final` scores by both `gender` and `term`:

```python
pd.crosstab(df['gender'], df['term'], values=df['final'], aggfunc='mean')
```

Output (average final scores):

term	fall
female	17.0
male	21.67

Pandas Data Structures

Data Visualization with Pandas

SPECIALIZATION

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
PANDAS AND NUMPY

LECTURE 1

PANDAS

Summary of Visualization Techniques

- **Line Plots**: Ideal for time series and trends.
- **Bar/Box/Scatter Plots**: Great for categorical comparisons and relationships between variables.
- **Histograms**: Useful for understanding data distribution.
- **Customization**: Add titles, labels, legends, and use different styles to make plots informative.
- **Advanced Customization**: Use Matplotlib for fine-tuning and Seaborn for aesthetically enhanced statistical plots.

Plotting Basics

1. Plotting Basics

a. Line Plots (`plot()`)

- **Line plots** are ideal for visualizing trends over time, and by default, Pandas uses line plots for time series data:

```python
df['Temperature'].plot(title='Temperature Over Time', ylabel='Temperature (°C)
plt.show()
```

Example Output:

- A simple line plot showing temperature changes over time.

Plotting Basics

b. Bar Plots (`bar()` and `barh()`)

- **Bar plots** are useful for categorical data. Pandas offers vertical and horizontal bar plots.
  - Vertical Bar Plot:

```python
df['Wind Speed'].plot(kind='bar', title='Wind Speed Over Time')
plt.show()
```

  - Horizontal Bar Plot:

```python
df['Wind Speed'].plot(kind='barh', title='Wind Speed Over Time')
plt.show()
```

Example Output:

- A bar plot showing wind speed across different times.

Plotting Basics

c. Histograms (`hist()`)

- **Histograms** show the distribution of a single variable, such as the temperature distribution:

```python
df['Temperature'].plot(kind='hist', bins=10, title='Temperature Distribution')
plt.show()
```

Example Output:

- A histogram showing how temperature values are distributed.

Plotting Basics

d. Box Plots (`box()`)

- **Box plots** are great for visualizing the spread of data, including quartiles and outliers:

```python
df[['Temperature', 'Wind Speed']].plot(kind='box', title='Temperature and Wind
plt.show()
```

Example Output:

- Box plots showing the distribution and outliers of temperature and wind speed.

Plotting Basics

e. Scatter Plots (`scatter()`)

- **Scatter plots** visualize the relationship between two variables:

```python
df.plot.scatter(x='Temperature', y='Wind Speed', title='Wind Speed vs Temperatu
plt.show()
```

Copy code

Example Output:

- A scatter plot to identify correlations between temperature and wind speed.

Customizing Plots

a. Titles and Labels

- Add **titles** and **axis labels** to make your plots more readable:

```python
df['Temperature'].plot(title='Temperature Over Time')
plt.xlabel('Time')
plt.ylabel('Temperature (°C)')
plt.show()
```

b. Legends

- For plots with multiple series, **legends** are automatically added but can be customized:

```python
df[['Temperature', 'Wind Speed']].plot(title='Temperature and Wind Speed')
plt.legend(['Temp', 'Wind Speed'])
plt.show()
```

Customizing Plots

c. Styles and Themes

- Pandas integrates Matplotlib **styles** and **themes** to make plots visually appealing. You can set the style globally:

```python
plt.style.use('ggplot')
df['Temperature'].plot(title='Temperature Over Time')
plt.show()
```

Copy code

Popular styles: `'ggplot'`, `'seaborn'`, `'fivethirtyeight'`.

Integration with Matplotlib and Seaborn

3. Integration with Matplotlib and Seaborn

a. Fine-Tuning with Matplotlib

- Pandas plots can be customized further with Matplotlib functions:

```python
ax = df['Temperature'].plot(title='Temperature Over Time')
ax.set_xlabel('Time')
ax.set_ylabel('Temperature (°C)')
plt.xticks(rotation=45)
plt.show()
```

Example: Customize tick labels, fonts, and more using Matplotlib commands after generating the Pandas plot.

b. Enhancing with Seaborn

- **Seaborn** offers advanced, aesthetically pleasing statistical plots. For example, a **heatmap**:

```python
import seaborn as sns
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.show()
```

Example Output: A correlation heatmap showing relationships between variables such as `Temperature`, `Wind Speed`, and `Humidity`.

Pandas Data Structures
Time Series Analysis in Pandas

SPECIALIZATION

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
PANDAS AND NUMPY

LECTURE 1

PANDAS

Time Series Analysis in Pandas

- **Working with Datetime**: Convert columns to `datetime` format to handle time-indexed data.
- **Resampling**: Use `resample()` to change the frequency of the time series data.
- **Rolling Window**: Use rolling calculations for moving averages and volatility analysis.
- **Shifting and Lagging**: Use `shift()` for autoregressive modeling or comparing current values with past data.
- **Time-Based Indexing**: Extract data for specific dates or ranges using `loc[]`.
- **Decomposing Time Series**: Break down the time series into trend, seasonal, and residual components.
- **Handling Missing Data**: Use `interpolate()` or fill missing timestamps with `asfreq()`.
- **Autocorrelation**: Detect seasonality or repeating patterns using `autocorr()` and autocorrelation plots.

Time Series Analysis in Pandas

Dataset Overview

Year	Month	Day	Hour	Temperature	Relative Humidity	Mean Sea Level Pressure	Total Precipitation	Wind Speed	Wind Direction
2018	11	6	0	27.02	58	1010.6	0	7.28	8.53
2018	11	6	1	26.73	58	1011.1	0	7.07	14.74
2018	11	6	2	26.38	58	1010.3	0	5.86	10.62
...	...	...	...	...	...	...	...	...	...

Working with Dates and Times

1. Working with Dates and Times

- **Converting to Datetime Format:**
  - Combine the `Year`, `Month`, `Day`, and `Hour` columns to create a **datetime** index:

```python
df['datetime'] = pd.to_datetime(df[['Year', 'Month', 'Day', 'Hour']])
df.set_index('datetime', inplace=True)
```

Before Conversion:

Year	Month	Day	Hour	Temperature	...
2018	11	6	0	27.02	...

After Conversion:

datetime	Temperature	...
2018-11-06 00:00:00	27.02	...
2018-11-06 01:00:00	26.73	...

Resampling and Frequency Conversion

2. Resampling and Frequency Conversion

- **Resampling Data**:
  - Resample the time series to daily frequency, taking the **mean** temperature for each day:

```python
df_daily = df.resample('D').mean()
```

Before Resampling (hourly data):

datetime	Temperature
2018-11-06 00:00:00	27.02
2018-11-06 01:00:00	26.73

After Resampling (daily mean):

datetime	Temperature
2018-11-06	28.15
2018-11-07	25.67

- **Converting Frequency**:
  - Convert the data to a **weekly** frequency, using the sum of precipitation:

```python
df_weekly = df.resample('W').sum()['Total Precipitation']
```

Rolling Window Calculations

3. Rolling Window Calculations

- **Applying a Rolling Window**:
  - Calculate the 3-hour rolling **mean** for temperature:

```python
df['rolling_temp'] = df['Temperature'].rolling(window=3).mean()
```

Example:

datetime	Temperature	rolling_temp
2018-11-06 00:00:00	27.02	NaN
2018-11-06 01:00:00	26.73	NaN
2018-11-06 02:00:00	26.38	26.71

- **Calculating Rolling Standard Deviation**:
  - Calculate the rolling standard deviation for `Wind Speed` with a window of 6 hours:

```python
df['rolling_wind_speed_std'] = df['Wind Speed'].rolling(window=6).std()
```

Rolling Window Calculations

3. Rolling Window Calculations

- **Applying a Rolling Window**:
  - Calculate the 3-hour rolling **mean** for temperature:

```python
df['rolling_temp'] = df['Temperature'].rolling(window=3).mean()
```

Example:

datetime	Temperature	rolling_temp
2018-11-06 00:00:00	27.02	NaN
2018-11-06 01:00:00	26.73	NaN
2018-11-06 02:00:00	26.38	26.71

- **Calculating Rolling Standard Deviation**:
  - Calculate the rolling standard deviation for `Wind Speed` with a window of 6 hours:

```python
df['rolling_wind_speed_std'] = df['Wind Speed'].rolling(window=6).std()
```

Shifting and Lagging Data

4. Shifting and Lagging Data

- **Shifting Time Series:**
  - Use `shift()` to **lag** or **lead** data, which is useful for comparing current values to past or future values.
  - Example: Shift the `Temperature` data forward by 1 hour:

```python
df['temp_shifted'] = df['Temperature'].shift(1)
```

Output:

datetime	Temperature	temp_shifted
2018-11-06 00:00:00	27.02	NaN
2018-11-06 01:00:00	26.73	27.02
2018-11-06 02:00:00	26.38	26.73

- **Lagged Data for Comparison:**
  - Useful for building **autoregressive models**, where future values depend on past values.

Time-Based Indexing and Slicing

5. Time-Based Indexing and Slicing

- **Slicing Data by Date**:
  - Select data for a specific time period using **time-based indexing**.
  - Example: Extract data only for November 6, 2018:

```python
df.loc['2018-11-06']
```

Output:

datetime	Temperature	...
2018-11-06 00:00:00	27.02	...
2018-11-06 01:00:00	26.73	...

- **Slicing a Range of Dates**:
  - Extract data between two specific dates:

```python
df.loc['2018-11-06':'2018-11-07']
```

Time Zone Handling

6. Time Zone Handling

- **Setting a Time Zone:**
  - You can **localize** a time series to a specific time zone using `tz_localize()`:

```python
df.index = df.index.tz_localize('UTC')
```

- **Converting Between Time Zones:**
  - Convert the time series from one time zone to another using `tz_convert()`:

```python
df.index = df.index.tz_convert('US/Eastern')
```

Decomposing Time Series

7. Decomposing Time Series

- **Decomposing a Time Series:**
  - Decompose a time series into **trend**, **seasonal**, and **residual** components using `seasonal_decompose()` (requires `statsmodels`).

python Copy code

```python
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(df['Temperature'], model='additive', period=
decomposition.plot()
plt.show()
```

This helps identify **seasonality**, **long-term trends**, and **noise** in the data.

Handling Missing Data in Time Series

8. Handling Missing Data in Time Series

- **Filling Missing Timestamps**:
  - Sometimes time series data has **gaps**. You can use `asfreq()` to fill missing timestamps with NaNs.

```python
df = df.asfreq('H')  # Set frequency to hourly, filling gaps with NaNs
```

- **Interpolating Missing Data**:
  - Use `interpolate()` to fill missing values in time series data, which is especially useful in sensor data or when values are missing for certain time intervals.

```python
df['Temperature'] = df['Temperature'].interpolate(method='linear')
```

Autocorrelation and Partial Autocorrelation

9. Autocorrelation and Partial Autocorrelation

- **Autocorrelation**:
  - Measure how the data correlates with itself at different time lags, using `autocorr()`. This is useful for detecting **seasonality**.

python Copy code

```python
df['Temperature'].autocorr(lag=1)  # Autocorrelation with a lag of 1
```

- **Visualizing Autocorrelation**:
  - Use `plot_acf()` from `statsmodels` to visualize autocorrelation.

python Copy code

```python
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(df['Temperature'], lags=24)
plt.show()
```

Forecasting with Time Series Models

10. Forecasting with Time Series Models

- **Simple Moving Average Forecast**:
  - Use rolling windows to predict future values based on historical averages.

```python
df['forecast'] = df['Temperature'].rolling(window=3).mean().shift(-1)
```

- **Advanced Time Series Models**:
  - Build models like **ARIMA** or **Exponential Smoothing** for forecasting future data trends (requires `statsmodels`).

Forecasting with Time Series Models

10. Forecasting with Time Series Models

- **Simple Moving Average Forecast**:
  - Use rolling windows to predict future values based on historical averages.

```python
df['forecast'] = df['Temperature'].rolling(window=3).mean().shift(-1)
```

- **Advanced Time Series Models**:
  - Build models like **ARIMA** or **Exponential Smoothing** for forecasting future data trends (requires `statsmodels`).

Visualizing Time Series Data with Pandas

Summary of Time Series Visualizations in Pandas

- **Line Plots**: Visualize trends and patterns over time.
- **Rolling Statistics**: Highlight long-term trends by smoothing short-term fluctuations.
- **Autocorrelation**: Detect repetitive patterns or seasonality in your time series.
- **Heatmaps**: Use Seaborn to show hourly/daily patterns visually.
- **Scatter Plots**: Explore relationships between different time series variables.

Line Plot for Time Series

1. Line Plot for Time Series

- **Basic Time Series Plot**:
  - A simple line plot to visualize time series trends over time:

```python
df['Temperature'].plot(title='Temperature Over Time', ylabel='Temperature (°C)
plt.show()
```

Example Output:

Description:

- A basic line plot is often the first visualization to observe trends and patterns in the data. This is ideal for showing how the `Temperature` changes over time.

Multiple Time Series Plot

2. Multiple Time Series Plot

- **Plotting Multiple Columns**:
  - You can plot multiple time series on the same graph. For example, plot both `Temperature` and `Wind Speed`:

```python
df[['Temperature', 'Wind Speed']].plot(title='Temperature and Wind Speed Over T
plt.show()
```

Copy code

Example Output:

- This plot allows you to compare multiple time series on the same axis, showing how different variables relate to each other over time.

Rolling Statistics Plot

2. Multiple Time Series Plot

- **Plotting Multiple Columns**:
  - You can plot multiple time series on the same graph. For example, plot both `Temperature` and `Wind Speed`:

```python
df[['Temperature', 'Wind Speed']].plot(title='Temperature and Wind Speed Over T
plt.show()
```

Example Output:

- This plot allows you to compare multiple time series on the same axis, showing how different variables relate to each other over time.

Time Series Resampling and Plotting

- **Daily Resampling and Plotting**:
  - Resample the data to a daily frequency and plot the resampled data:

```python
df_daily = df['Temperature'].resample('D').mean()
df_daily.plot(title='Daily Average Temperature', figsize=(10, 6))
plt.show()
```

Example Output:

- This plot shows how the average daily temperature changes over time, providing insights into broader trends than hourly fluctuations.

Visualizing Missing Data

5. Visualizing Missing Data

- **Plotting Missing Data**:
  - Visualize missing data by creating a binary column indicating whether data is missing:

```python
df['Temperature_missing'] = df['Temperature'].isnull().astype(int)
df['Temperature_missing'].plot(kind='bar', title='Missing Temperature Data', fi
plt.show()
```

Example Output:

- This bar plot highlights where missing data occurs in the dataset. It is useful when you want to visualize data quality over time.

Autocorrelation Plot

6. Autocorrelation Plot

- **Autocorrelation Visualization**:
  - An autocorrelation plot shows the correlation of a time series with its own lagged values. This helps in identifying seasonality or repetitive patterns.

```python
from pandas.plotting import autocorrelation_plot
autocorrelation_plot(df['Temperature'])
plt.title('Autocorrelation of Temperature')
plt.show()
```

Example Output:

- The plot shows how the temperature correlates with its own past values (lag). Peaks at specific intervals can suggest seasonality.

Time Series Heatmap (Matplotlib/Seaborn)

7. Time Series Heatmap (Matplotlib/Seaborn)

- **Heatmap of Hourly Data**:
  - Although not directly supported in Pandas, you can easily create a heatmap using Seaborn by pivoting the data:

python Copy code

```
import seaborn as sns
df_pivot = df.pivot_table(index='Day', columns='Hour', values='Temperature')
sns.heatmap(df_pivot, cmap='coolwarm')
plt.title('Temperature Heatmap by Day and Hour')
plt.show()
```

Example Output:

- This heatmap visualizes the temperature across different hours and days, making it easy to spot periods of high or low temperature.

Scatter Plot for Time Series

8. Scatter Plot for Time Series

- **Scatter Plot with Time on the X-axis:**
  - If you want to visualize relationships between two time series (e.g., `Temperature` and `Wind Speed`), use a scatter plot:

```python
df.plot.scatter(x='Temperature', y='Wind Speed', title='Wind Speed vs Temperatu
plt.show()
```

Copy code

Example Output:

- This scatter plot shows the relationship between temperature and wind speed, which might reveal correlations or patterns between the two variables.

Pandas Data Structures
Advanced Topics

SPECIALIZATION

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
PANDAS AND NUMPY

LECTURE 1

PANDAS

Summary of Advanced Pandas Techniques

- **Merging and Joining**: Combine DataFrames using `concat()`, `merge()`, and `join()`.
- **Pivot Tables and Reshaping**: Use `pivot_table()` for summaries, and `melt()` or `stack()` to reshape data.
- **MultiIndexing**: Handle hierarchical indexes and multi-level access.
- **Large Datasets**: Use `chunksize` for memory-efficient data reading and optimize memory usage with data type conversions.
- **Performance Optimization**: Leverage vectorization and categorical data types for faster operations and reduced memory footprint.

Merging and Joining DataFrames

1. Merging and Joining DataFrames

a. Concatenation (`concat()`)

- **Concatenation** combines DataFrames either vertically (row-wise) or horizontally (column-wise):

```python
pd.concat([df1, df2], axis=0)  # Vertical concatenation (stacking)
pd.concat([df1, df2], axis=1)  # Horizontal concatenation (side by side)
```

Copy code

Example Output:

- Vertically stacks two DataFrames with the same columns.

Merging and Joining DataFrames

b. Merging on Keys (`merge()`)

- **Merging** combines DataFrames based on common columns or keys, similar to SQL joins:

```python
pd.merge(df1, df2, on='key_column', how='inner')  # Inner join
pd.merge(df1, df2, on='key_column', how='outer')  # Outer join
```

Example Output:

- Merges two DataFrames based on a shared key column, performing different types of joins (inner, outer, left, right).

Merging and Joining DataFrames

c. Joining on Indexes (`join()`)

- **Join** combines DataFrames based on their indexes:

```python
df1.join(df2, how='left')  # Joins on index, left join by default
```

Example Output:

- Combines two DataFrames using their indexes for matching rows.

Pivot Tables and Reshaping

2. Pivot Tables and Reshaping

a. Creating Pivot Tables (`pivot_table()`)

- **Pivot tables** summarize data and allow you to aggregate values across different dimensions:

python Copy code

```
df.pivot_table(values='final', index='gender', columns='year', aggfunc='mean')
```

Example Output:

- A pivot table showing the mean `final` score by `gender` and `year`.

Pivot Tables and Reshaping

b. Melting and Stacking

- **Melting** reshapes data by converting columns into rows:

```python
pd.melt(df, id_vars=['gender'], value_vars=['Temperature', 'Wind Speed'])
```

Example Output:

- Converts wide-format data to long-format by unpivoting columns into rows.

- **Stacking/Unstacking** converts a DataFrame from wide to long format and vice versa:

```python
df.stack()  # Converts columns into index levels
df.unstack()  # Converts index levels into columns
```

Multi-Indexing

3. MultiIndexing

a. Hierarchical Indexing

- **Hierarchical (Multi) Indexing** allows you to set multiple levels of indexes:

```python
df.set_index(['year', 'month', 'day'], inplace=True)
```

Example Output:

- Creates a DataFrame where `year`, `month`, and `day` are used as hierarchical indexes.

b. Indexing with Multiple Levels

- **Accessing Data with MultiIndex**:
  - Select data at specific levels:

```python
df.loc[('2018', '11', '06')]
```

Example Output:

- Extracts rows corresponding to a specific date using hierarchical index levels.

Working with Large Datasets

4. Working with Large Datasets

a. Reading Data in Chunks

- Reading Large Files in Chunks:
  - For large datasets, use `chunksize` to read data in chunks:

```python
chunk_iter = pd.read_csv('large_data.csv', chunksize=10000)
for chunk in chunk_iter:
    process(chunk)  # Process each chunk independently
```

Copy code

Example Output:

- Efficiently reads large datasets by loading smaller chunks into memory.

Working with Large Datasets

b. Memory Optimization Techniques

- **Optimizing Memory Usage**:
  - Convert data types to optimize memory:

```python
df['column'] = df['column'].astype('float32')  # Use smaller data types
```

  - Use `memory_usage()` to check how much memory the DataFrame consumes:

```python
df.memory_usage(deep=True)
```

Example Output:

- Memory usage is reduced by converting columns to more efficient data types.

Performance Optimization

5. Performance Optimization

a. Vectorization

- **Vectorization** allows operations to be applied to entire arrays without looping:

```python
df['new_column'] = df['Temperature'] * df['Wind Speed']  # Element-wise multip
```

Example Output:

- Operations are applied across the DataFrame much faster than with explicit loops.

b. Using Categorical Data Types

- **Categorical Data Types** optimize memory usage for columns with a limited number of distinct values:

```python
df['gender'] = df['gender'].astype('category')
```

Example Output:

- The memory usage for `gender` is significantly reduced by using the `category` type.

جامعة الامير سلطان
PRINCE SULTAN UNIVERSITY

FOR

SPECIALIZATION

Robotics & Internet-of-Things

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 3
Pandas and Numpy Data Structures

LECTURE 2
Pandas Data Analytics
Case Study

Prof. Anis Koubaa

SEP 2024