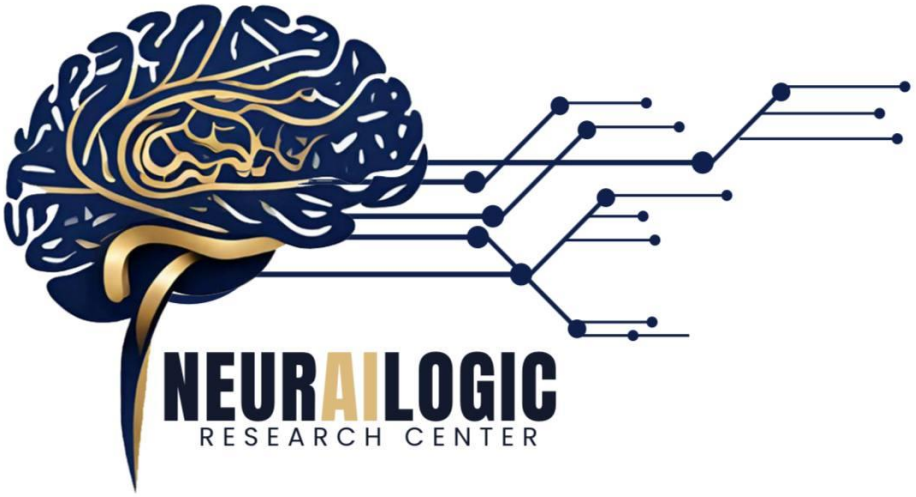

EEG Processing:

An Entry to the World of Brain Waves

Session 5 – Part 1:

Python – Numpy

Author: Mohammadreza Shahsavari

Contact: mohamadrezashahsavary@gmail.com

What you are going to learn in this session?

Welcome to the session on EEG metadata processing with Python! In this session, you'll learn how to use the powerful Pandas library to handle and analyze EEG data. We'll start by understanding the importance of Pandas in data analysis and EEG processing, and then move on to exploring its primary data structures: Series and DataFrames. You'll learn how to inspect, clean, and manipulate your data, and how to load data from CSV and Excel files. We'll also cover data transformation techniques, including applying functions, grouping, and aggregating data. By the end of this session, you'll have the skills to efficiently manage and analyze EEG datasets. Let's dive in!

- *Mohammadreza Shahsavari*

Table of Contents

1. Introduction to Pandas

Overview

Pandas is a powerful open-source data analysis and manipulation library for Python. It provides data structures and functions needed to work with structured data seamlessly and intuitively. The two primary data structures in Pandas are Series (one-dimensional) and DataFrame (two-dimensional), which allow for easy handling of a wide range of data types and formats.

Importance of Pandas in Data Analysis and EEG Processing

In EEG processing projects, Pandas is particularly useful because EEG datasets often include not only the raw EEG signals but also metadata and event information. Metadata might include patient details like age, gender, or medical history, while event information can include timestamps and descriptions of stimuli or activities during EEG recording sessions. These additional data types are often stored in Excel or CSV files and require robust handling and integration with the primary EEG data. Pandas provides the tools to efficiently load, clean, manipulate, and analyze this supplementary data, enabling comprehensive analysis and better insights.

Installing Pandas Using pip And Importing It in Python

To install Pandas, you can use Python's package manager, pip. Open your command line interface and execute the following command:

```
pip install pandas
```

This command will download and install the latest version of Pandas and its dependencies.

Once installed, you can import Pandas into your Python scripts using the following import statement:

```
import pandas as pd
```

This convention (pd) is widely used in the Python community and helps keep the code concise and readable.

2. Data Structures in Pandas: Series and DataFrames

Pandas, a powerful data manipulation library in Python, provides two primary data structures: **Series** and **DataFrames**. These structures are designed to handle and manipulate data efficiently, making them essential tools for data analysis, including the handling of metadata in EEG processing.

Series

A **Series** is a one-dimensional array-like object that can hold data of any type, such as integers, floats, strings, and even Python objects. It is similar to a column in a table or a list in Python, but with additional capabilities.

Key features of a Series:

- **Indexing**: Each element in a Series is associated with an index, allowing for label-based data retrieval. This index can be customized and need not be unique or ordered.
- **Homogeneous Data**: A Series holds homogeneous data, meaning all elements are of the same data type.
- **Arithmetic Operations**: Series support element-wise operations and can align data with different indexes, making mathematical operations straightforward.

Example of creating a Series with subject ages:

```
import pandas as pd

# Creating a Series for subject ages
subject_ages = pd.Series([29, 35, 42, 58], index=['Subject1', 'Subject2',
'Subject3', 'Subject4'])
print(subject_ages)
```

Output:

```
Subject1    29
Subject2    35
Subject3    42
Subject4    58
dtype: int64
```

Inspecting and Manipulating a Series:

```
# Accessing a single element by index
print(subject_ages['Subject2'])  # Output: 35

# Accessing multiple elements
print(subject_ages[['Subject1', 'Subject3']])

# Performing arithmetic operations
subject_ages += 1
print(subject_ages)
```

Output:

```
Subject1    29
Subject3    42
dtype: int64

Subject1    30
Subject2    36
Subject3    43
Subject4    59
dtype: int64
```

DataFrame

A **DataFrame** is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns). It is similar to a spreadsheet or a SQL table, and it can contain different data types in different columns.

Key features of a DataFrame:

- **Tabular Data**: DataFrames are structured as tables with rows and columns, where each column can be of a different data type.
- **Indexing**: Both rows and columns in a DataFrame have indices, allowing for versatile data selection and manipulation.
- **Alignment and Operations**: DataFrames support arithmetic operations on rows and columns, with automatic data alignment based on row and column labels.
- **Flexible Data Input**: DataFrames can be created from various data sources such as lists, dictionaries, Series, NumPy arrays, and even other DataFrames.

Example of creating a DataFrame with subject metadata:

```
import pandas as pd

# Creating a DataFrame for subject metadata
data = {
    'SubjectID': ['S1', 'S2', 'S3', 'S4'],
    'Age': [29, 35, 42, 58],
    'Gender': ['F', 'M', 'F', 'M'],
    'Handedness': ['Right', 'Left', 'Right', 'Right']
}

subject_metadata = pd.DataFrame(data)
print(subject_metadata)
```

Output:

```
  SubjectID  Age Gender Handedness
0        S1   29      F      Right
1        S2   35      M       Left
2        S3   42      F      Right
3        S4   58      M      Right
```

3. Data Exploration

Exploring your data is a crucial step in understanding its structure, contents, and the types of analysis you can perform. Pandas offers several built-in functions to facilitate this process.

Inspecting Data with head(), tail(), info(), and describe()

1. head()

The head() function allows you to view the first few rows of your DataFrame. By default, it shows the first five rows, but you can specify the number of rows to display.

```python
import pandas as pd

# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Age': [25, 30, 35, 40, 45],
    'Gender': ['F', 'M', 'M', 'M', 'F'],
    'Score': [85, 88, 90, 85, 87]
}
df = pd.DataFrame(data)

# Displaying the first few rows
print("First few rows of the DataFrame:")
print(df.head())
```

Output:

```
First few rows of the DataFrame:
      Name  Age Gender  Score
0    Alice   25      F     85
1      Bob   30      M     88
2  Charlie   35      M     90
3    David   40      M     85
4      Eva   45      F     87
```

2. tail()

The tail() function allows you to view the last few rows of your DataFrame. By default, it shows the last five rows, but you can specify the number of rows to display.

```
# Displaying the last few rows
print("Last few rows of the DataFrame:")
print(df.tail(3))
```

Output:

```
Last few rows of the DataFrame:
      Name  Age Gender  Score
2  Charlie   35      M     90
3    David   40      M     85
4      Eva   45      F     87
```

3. info()

The info() function provides a concise summary of the DataFrame, including the index dtype, column dtypes, non-null values, and memory usage.

```
# Getting a concise summary of the DataFrame
print("Summary of the DataFrame:")
print(df.info())
```

Output:

```
Summary of the DataFrame:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    5 non-null      object
 1   Age     5 non-null      int64
 2   Gender  5 non-null      object
 3   Score   5 non-null      int64
dtypes: int64(2), object(2)
memory usage: 288.0+ bytes
None
```

4. describe()

The describe() function generates descriptive statistics of the numerical columns in the DataFrame, such as count, mean, standard deviation, min, max, and quartiles.

```
# Getting summary statistics of the DataFrame
print("Descriptive statistics of the DataFrame:")
print(df.describe())
```

Output:

```
Descriptive statistics of the DataFrame:
             Age      Score
count   5.000000   5.000000
mean   35.000000  87.000000
std     7.905694   2.236068
min    25.000000  85.000000
25%    30.000000  85.000000
50%    35.000000  87.000000
75%    40.000000  88.000000
max    45.000000  90.000000
```

Accessing Columns, Rows, and Cells with iloc and loc

1. Accessing Columns

In a DataFrame, columns represent a specific attribute or feature of the data. You can access these columns by referencing their names directly. This is useful for data exploration and analysis, allowing you to focus on specific attributes without modifying the underlying data.

```
# Accessing a single column
print("Accessing the 'Name' column:")
print(df['Name'])

# Accessing multiple columns
print("Accessing the 'Name' and 'Score' columns:")
print(df[['Name', 'Score']])
```

Output:

```
Accessing the 'Name' column:
0      Alice
1        Bob
2    Charlie
3      David
4        Eva
Name: Name, dtype: object

Accessing the 'Name' and 'Score' columns:
      Name  Score
0    Alice     85
1      Bob     88
2  Charlie     90
3    David     85
4      Eva     87
```

Here, first we retrieve the 'Name' column, which gives us a series of names from the DataFrame. Next, we access two columns, 'Name' and 'Score', which returns a DataFrame containing only these two columns.

2. Accessing Rows by Integer Location with iloc

The iloc function is used for accessing rows by their numerical index, making it ideal when you need to work with rows based on their position in the DataFrame. This is particularly useful for slicing and selecting specific parts of the DataFrame without relying on row labels.

```
# Accessing a single row by its integer index
print("Accessing the second row:")
print(df.iloc[1])

# Accessing multiple rows by their integer indices
print("Accessing the first and third rows:")
print(df.iloc[[0, 2]])
```

Output:

```
Accessing the second row:
Name       Bob
Age         30
Gender       M
Score       88
Name: 1, dtype: object

Accessing the first and third rows:
      Name  Age Gender  Score
0    Alice   25      F     85
2  Charlie   35      M     90
```

Here, first we access the second row (index 1), which returns all columns for that row. Then we demonstrated accessing the first and third rows (index 0 and 2), returning a DataFrame with the specified rows.

3. Accessing Rows and Columns by Labels with loc

The loc function is used for accessing rows and columns by their labels, providing a more intuitive way to work with data when you know the exact labels of the rows and columns you need. This is useful for data manipulation and extraction based on meaningful labels.

```
# Accessing a single cell by its row and column labels
print("Accessing the cell at row 1 and column 'Age':")
print(df.loc[1, 'Age'])

# Accessing a row and specific columns by their labels
print("Accessing the second row with 'Name' and 'Score' columns:")
print(df.loc[1, ['Name', 'Score']])
```

Output:

```
Accessing the cell at row 1 and column 'Age':
30

Accessing the second row with 'Name' and 'Score' columns:
Name      Bob
Score      88
Name: 1, dtype: object
```

In this example, we first access the cell located at row label 1 and column 'Age', retrieving the age value for that specific row. Then we accessed the second row by its label (1) and only the 'Name' and 'Score' columns, returning a series with the specified labels and values.

4. Data Loading

In EEG processing projects, metadata such as patient information (age, gender, etc.) and event information (timestamps, stimuli details) are often stored in CSV or Excel (xlsx) files. In some cases, the EEG signal data itself may also be saved in these formats. Pandas provides powerful functions to read these files into DataFrames for analysis.

Reading Data from CSV Files using pd.read_csv()

CSV (Comma Separated Values) files are a common format for storing tabular data. The pd.read_csv() function reads a CSV file into a DataFrame.

```
import pandas as pd

# Example CSV content:
# patient_id,age,gender,condition
# 1,25,F,Healthy
# 2,30,M,Epileptic
# 3,28,F,Healthy

# Reading data from a CSV file
df_csv = pd.read_csv('eeg_metadata.csv')
print("Data read from CSV file:")
print(df_csv)
```

Output:

```
Data read from CSV file:
   patient_id  age gender  condition
0           1   25      F    Healthy
1           2   30      M  Epileptic
2           3   28      F    Healthy
```

Reading Data from Excel Files using pd.read_excel()

Excel files (xlsx) are widely used for storing tabular data. The pd.read_excel() function reads an Excel file into a DataFrame. It can also handle multiple sheets within the same file.

```
# Reading data from an Excel file
df_excel = pd.read_excel('eeg_metadata.xlsx', sheet_name='Sheet1')
print("Data read from Excel file:")
print(df_excel)
```

Output:

```
Data read from Excel file:
   patient_id  age gender  condition
0           1   25      F    Healthy
1           2   30      M  Epileptic
2           3   28      F    Healthy
```

By using pd.read_csv() and pd.read_excel(), you can efficiently load EEG metadata, event information, and even EEG signal data from CSV and Excel files into Pandas DataFrames. This capability is crucial for managing and analyzing the diverse types of data involved in EEG processing projects.

5. Cleaning and Manipulating Data

Handling Missing Values with isna() and fillna()

In real-world datasets, it's common to encounter missing values in data frames. For instance, consider an Excel file where some cells are left empty. When you load this file into a Pandas DataFrame, these empty cells are represented as NaN values. Pandas provides robust tools to identify and handle these missing values effectively.

Identifying Missing Values

The isna() function helps identify missing values in a DataFrame. It returns a DataFrame of the same shape with boolean values indicating the presence of missing data.

```python
import pandas as pd

# Sample DataFrame with missing values
data = {
    'patient_id': [1, 2, 3, 4],
    'age': [25, 30, None, 45],
    'gender': ['F', 'M', 'F', None],
    'condition': ['Healthy', 'Epileptic', 'Healthy', 'Healthy']
}
df = pd.DataFrame(data)
```

```
# Identifying missing values
print("Identifying missing values:")
print(df.isna())
```

Output:

```
Identifying missing values:
   patient_id    age  gender  condition
0       False  False   False      False
1       False  False   False      False
2       False   True   False      False
3       False  False    True      False
```

Filling Missing Values

The fillna() function fills missing values with a specified value or method (such as mean, median, or mode).

```
# Filling missing values with a specific value
df_filled = df.fillna({
    'age': df['age'].mean(),  # Fill missing ages with the mean age
    'gender': 'Unknown'       # Fill missing genders with 'Unknown'
})

print("Data after filling missing values:")
print(df_filled)
```

Output:

```
Data after filling missing values:
   patient_id   age   gender  condition
0           1  25.0        F    Healthy
1           2  30.0        M  Epileptic
2           3  33.333333        F    Healthy
3           4  45.0  Unknown    Healthy
```

Converting Data Types with astype()

Sometimes, you need to convert the data type of a column to perform certain operations or to ensure consistency.

```
# Sample DataFrame with incorrect data types
data = {
    'patient_id': ['1', '2', '3', '4'],
    'age': ['25', '30', '35', '40'],
    'gender': ['F', 'M', 'F', 'M']
}
df = pd.DataFrame(data)

# Converting data types
df['patient_id'] = df['patient_id'].astype(int)
df['age'] = df['age'].astype(float)

print("Data after converting data types:")
print(df)
print("Data types:")
print(df.dtypes)
```

Output:

```
Data after converting data types:
   patient_id   age gender
0           1  25.0      F
1           2  30.0      M
2           3  35.0      F
3           4  40.0      M
Data types:
patient_id      int64
age           float64
gender         object
dtype: object
```

5. Data Transformation

Applying Functions with apply(), map(), applymap()

Pandas allows you to apply functions to DataFrame elements easily.

apply()

The apply() function applies a function along an axis of the DataFrame (either rows or columns).

```
# Applying a function to a column
df['age_squared'] = df['age'].apply(lambda x: x ** 2)
print("Data after applying function with apply():")
print(df)
```

Output:

```
Data after applying function with apply():
   patient_id   age gender  age_squared
0           1  25.0      F        625.0
1           2  30.0      M        900.0
2           3  35.0      F       1225.0
3           4  40.0      M       1600.0
```

map()

The map() function is used to map values from two series having one similar column. For mapping two series, the last column of the first should be the same as the index column of the second series, also the values should be unique.

```
# Mapping values in a column
df['gender_mapped'] = df['gender'].map({'F': 0, 'M': 1})
print("Data after mapping values with map():")
print(df)
```

Output:

```
Data after mapping values with map():
   patient_id   age gender  age_squared  gender_mapped
0           1  25.0      F        625.0              0
1           2  30.0      M        900.0              1
2           3  35.0      F       1225.0              0
3           4  40.0      M       1600.0              1
```

applymap()

The applymap() function applies a function to each element of the DataFrame.

```
# Applying a function to each element of the DataFrame
df[['age', 'age_squared']] = df[['age', 'age_squared']].applymap(lambda x: x / 2)
print("Data after applying function with applymap():")
print(df)
```

Output:

```
Data after applying function with applymap():
   patient_id   age gender  age_squared  gender_mapped
0           1  12.5      F        312.5              0
1           2  15.0      M        450.0              1
2           3  17.5      F        612.5              0
3           4  20.0      M        800.0              1
```

Grouping and Aggregating Data with groupby() and agg()

Grouping and aggregating data are powerful techniques for summarizing and analyzing data. These methods are particularly useful when you need to perform calculations on subsets of your data, such as calculating the average age or average score for each gender in an EEG dataset.

groupby() and agg()

The groupby() function groups the DataFrame by one or more columns, and the agg() function applies one or more aggregation functions to the grouped data.

Let's break down the process step-by-step using an example DataFrame.

```
import pandas as pd

# Sample DataFrame
data = {
    'patient_id': [1, 2, 3, 4, 5, 6],
    'age': [25, 30, 35, 40, 45, 50],
    'gender': ['F', 'M', 'F', 'M', 'F', 'M'],
    'score': [85, 88, 90, 85, 87, 89]
}
df = pd.DataFrame(data)

# Display the DataFrame
print("Original DataFrame:")
print(df)
```

Output:

```
Original DataFrame:
   patient_id  age gender  score
0           1   25      F     85
1           2   30      M     88
2           3   35      F     90
3           4   40      M     85
4           5   45      F     87
5           6   50      M     89
```

Step 1: Grouping by Gender

The groupby() function groups the DataFrame by the specified column, in this case, 'gender'. This means that all rows with the same gender will be grouped together.

```
# Grouping by gender
grouped = df.groupby('gender')
print("Grouped DataFrame:")
for name, group in grouped:
    print(f"\nGroup: {name}")
    print(group)
```

Output:

```
Grouped DataFrame:

Group: F
   patient_id  age gender  score
0           1   25      F     85
2           3   35      F     90
4           5   45      F     87

Group: M
   patient_id  age gender  score
1           2   30      M     88
3           4   40      M     85
5           6   50      M     89
```

Step 2: Aggregating Data

The agg() function applies one or more aggregation functions to each group. Here, we calculate the mean age and score for each gender group.

```
# Calculating mean age and score for each gender
aggregated = grouped.agg({'age': 'mean', 'score': 'mean'})
print("Grouped and aggregated data:")
print(aggregated)
```

Output:

```
Grouped and aggregated data:
         age  score
gender
F       35.0   87.33
M       40.0   87.33
```

This output shows that the average age for females is 35 years, and for males, it is 40 years. Similarly, the average score for both genders is 87.33.

Iterating Over DataFrame Rows

Sometimes, you might need to iterate over the rows of a DataFrame. While it's generally more efficient to use vectorized operations, Pandas provides methods for iteration when necessary.

Using iterrows()

The iterrows() function allows you to iterate over DataFrame rows as (index, Series) pairs. This can be useful for performing row-wise operations that are not easily vectorizable.

```
# Iterating over DataFrame rows
print("Iterating over DataFrame rows with iterrows():")
for index, row in df.iterrows():
    print(f"Index: {index}, Patient ID: {row['patient_id']}, Age: {row['age']}")
```

Output:

```
Iterating over DataFrame rows with iterrows():
Index: 0, Patient ID: 1, Age: 25
Index: 1, Patient ID: 2, Age: 30
Index: 2, Patient ID: 3, Age: 35
Index: 3, Patient ID: 4, Age: 40
Index: 4, Patient ID: 5, Age: 45
Index: 5, Patient ID: 6, Age: 50
```

Using apply() for Row-wise Operations

For row-wise operations, apply() can be more efficient than iterrows(). It allows you to apply a function to each row or column of the DataFrame.

Example: Creating an Age Category

Suppose we want to create a new column that categorizes patients as 'Young' if they are under 40 years old and 'Old' otherwise.

```
# Applying a function row-wise
df['age_category'] = df.apply(lambda row: 'Young' if row['age'] < 40 else 'Old',
axis=1)
print("Data after applying row-wise function with apply():")
print(df)
```

Output:

```
Data after applying row-wise function with apply():
   patient_id  age gender  score age_category
0           1   25      F     85        Young
1           2   30      M     88        Young
2           3   35      F     90        Young
3           4   40      M     85          Old
4           5   45      F     87          Old
5           6   50      M     89          Old
```