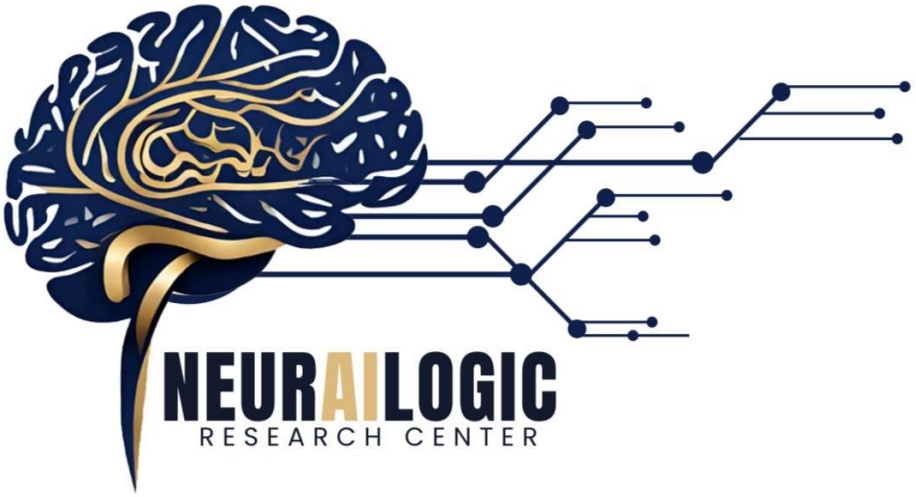


EEG Processing:

An Entry to the World of Brain Waves

Session 6:

Python – Matplotlib

Author: Mohammadreza Shahsavari

Contact: mohamadrezashahsavary@gmail.com

What you are going to learn in this session?

Welcome to the session on Matplotlib! Today, we'll dive into the world of data visualization using one of Python's most popular libraries. We'll start with the basics, showing you how to create simple plots and customize them with titles, labels, and legends. You'll get hands-on experience with different types of plots like line plots, scatter plots, bar charts, and histograms.

As we progress, we'll explore advanced customization techniques to enhance the aesthetics of your plots using styles, themes, and annotations. You'll learn how to make your plots more readable by customizing tick marks and grid lines. We'll also cover how to create multiple plots in a single figure using subplots and GridSpec for more complex layouts.

To top it off, we'll introduce you to interactive plotting with Matplotlib widgets, allowing you to create dynamic and interactive visualizations. By the end of this session, you'll have the skills to create beautiful, informative, and interactive plots to visualize your data effectively. Let's get started and unleash the power of Matplotlib together!

- *Mohammadreza Shahsavari*

Table of Contents

1. Introduction to Matplotlib

Overview

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It was originally designed to emulate MATLAB's plotting capabilities, making it a popular choice for scientists and engineers who were transitioning from MATLAB to Python. Matplotlib is highly customizable, allowing users to control every aspect of a figure and producing publication-quality graphics.

Matplotlib consists of several modules, but the most commonly used one is pyplot, which provides a MATLAB-like interface. This interface makes it easy to create plots by specifying only the essential components and defaulting the rest.

Importance of Matplotlib in Data Visualization

Data visualization is a critical component of data analysis, providing a way to see and understand trends, patterns, and outliers in data. Matplotlib's importance in data visualization stems from its versatility and integration with other libraries like NumPy and Pandas, making it ideal for plotting complex data such as EEG signals and demographic data.

For EEG processing, visualizing the data is essential for understanding the characteristics of the signals, identifying artifacts, and communicating findings. Matplotlib enables the creation of detailed and informative plots that can highlight key aspects of the data. Similarly, demographic data can be visualized to reveal distributions, correlations, and trends, facilitating better insights and decisions.

Installing Matplotlib Using pip and Importing It into Python

Installing Matplotlib is straightforward using Python's package manager, pip. Open your command line interface and execute the following command:

```
pip install matplotlib
```

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

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

```
import matplotlib.pyplot as plt
```

This import statement is widely used in the Python community and helps keep the code concise and readable. The `plt` alias is a common convention and is used throughout Matplotlib documentation and examples.

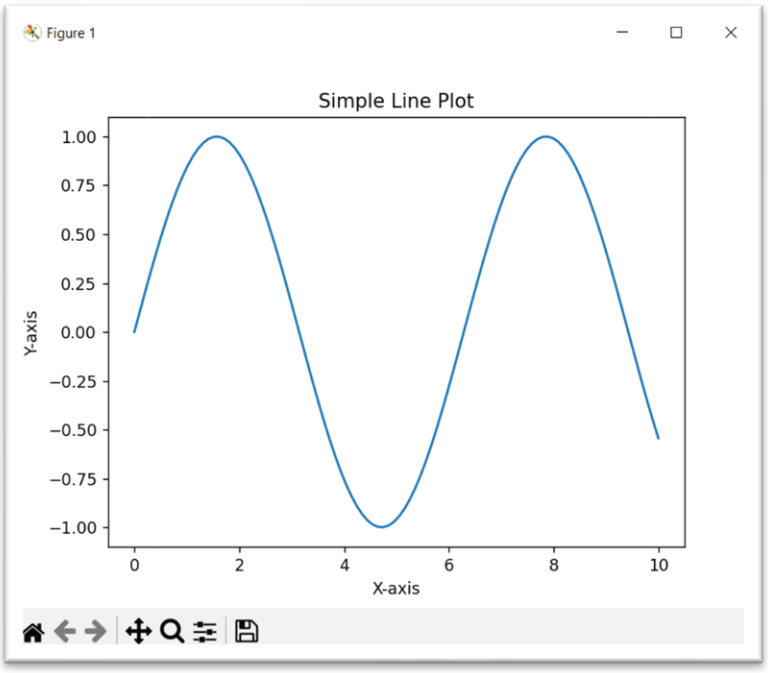
2. Basic Plotting with Matplotlib

Creating Simple Plots with plot()

The `plot()` function is the most basic and versatile plotting function in Matplotlib. It can create a wide range of plots, but it's primarily used for line plots.

```
import matplotlib.pyplot as plt
import numpy as np

# Example data
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Creating a simple line plot
plt.plot(x, y)
plt.title("Simple Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

Output:



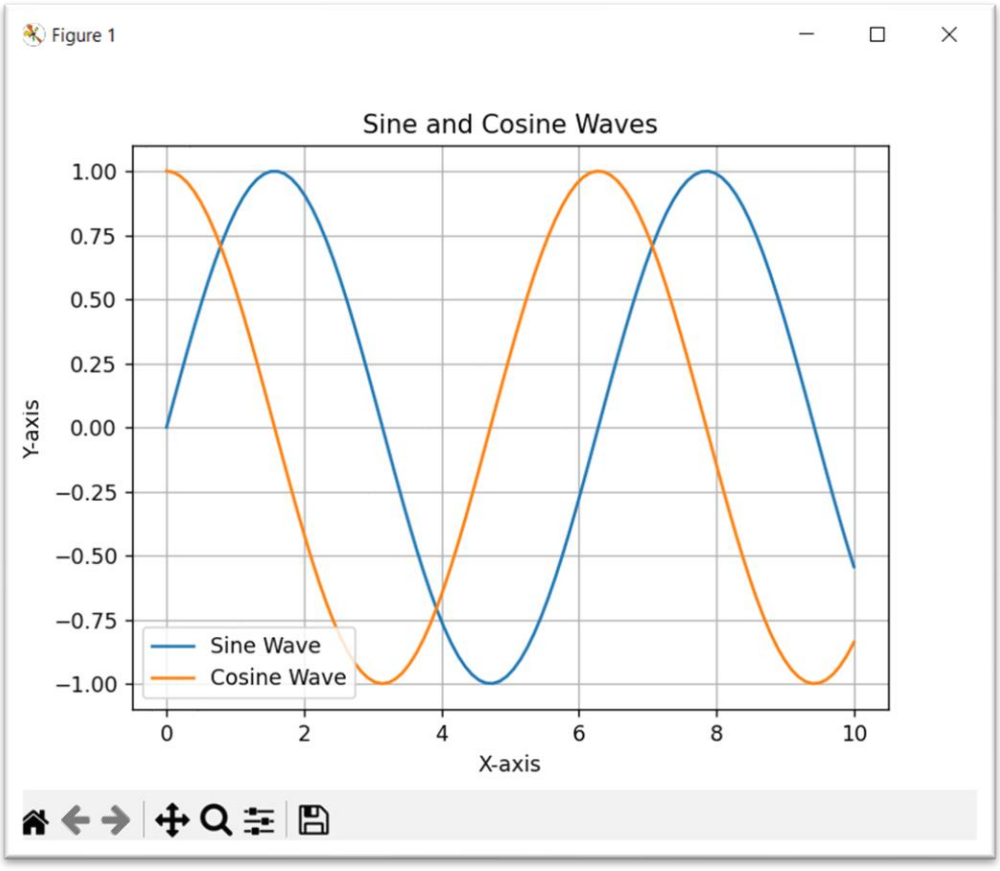
In this example, `numpy` is used to generate the data for the plot. The `plt.plot()` function creates a line plot, and the `plt.title()`, `plt.xlabel()`, and `plt.ylabel()` functions add a title and labels to the axes. Finally, `plt.show()` displays the plot.

Customizing Plots with Titles, Labels, and Legends

Customizing plots in Matplotlib is essential for making them informative and publication-ready. You can add titles, axis labels, and legends to make your plots more understandable.

```
# Example data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Creating a customized plot
plt.plot(x, y1, label="Sine Wave")
plt.plot(x, y2, label="Cosine Wave")
plt.title("Sine and Cosine Waves")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.legend()
plt.grid(True)
plt.show()
```

Output:



Plotting the Data:

- `plt.plot(x, y1, label="Sine Wave")` plots the sine wave and assigns the label "Sine Wave".
- `plt.plot(x, y2, label="Cosine Wave")` plots the cosine wave and assigns the label "Cosine Wave".

Adding a Title:

- `plt.title("Sine and Cosine Waves")` adds the title "Sine and Cosine Waves" to the plot, providing a concise description of the data being visualized.

Labeling the Axes:

- `plt.xlabel("X-axis")` labels the x-axis as "X-axis".
- `plt.ylabel("Y-axis")` labels the y-axis as "Y-axis". These labels help the viewer understand what each axis represents.

Adding a Legend:

- `plt.legend()` displays a legend on the plot, which uses the labels provided in the label parameter of the `plt.plot` function. This legend helps differentiate between the sine and cosine waves.

Adding a Grid:

- `plt.grid(True)` adds a grid to the plot, which can make it easier to read and interpret the data by providing reference lines.

Displaying the Plot:

- `plt.show()` renders and displays the plot.

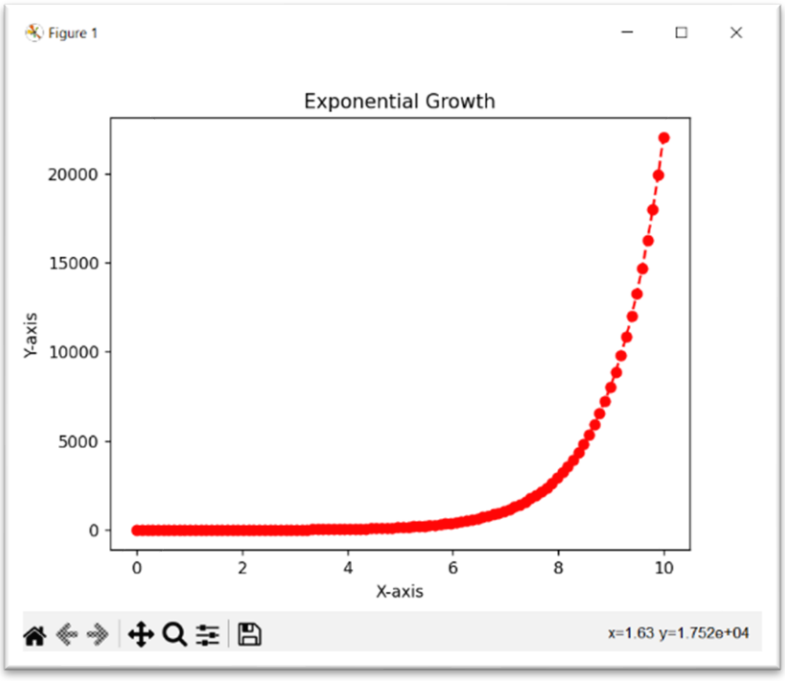
Types of Plots

Line Plots

We have encountered this type of plot in the previous section. Line plots are particularly useful for visualizing data trends over time or continuous variables. They can be created using the `plot()` function. Here is another example of a line plot, this time visualizing an exponential function (i.e. a series of values stored in an array following an exponential pattern).

```
# Example data
x = np.linspace(0, 10, 100)
y = np.exp(x)

# Creating a line plot
plt.plot(x, y, color='r', linestyle='--', marker='o')
plt.title("Exponential Growth")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

Output:



This will display a line plot of the exponential function with red dashed lines and circle markers. The appearance of the plot is determined by the following parameters in the code:

- `color='r'`: This sets the color of the line to red.
- `linestyle='--'`: This specifies that the line should be dashed.
- `marker='o'`: This adds circular markers at each data point.

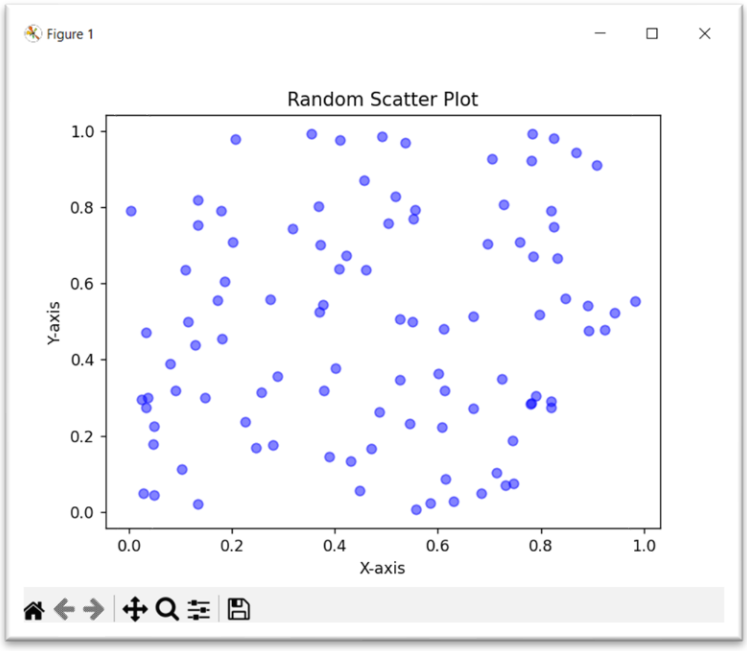
These parameters in the `plt.plot()` function customize the visual style of the plot, making the exponential growth pattern more distinguishable. The red color ensures the plot stands out, the dashed line style helps differentiate this plot from others that might use solid lines, and the circle markers highlight individual data points, providing a clear view of the rapid increase inherent to exponential functions. These stylistic choices are essential for making the plot both informative and visually engaging.

Scatter plots

Scatter plots are utilized to visualize the relationship between two continuous variables. They provide a clear picture of how one variable might correlate with another by displaying data points on a two-dimensional graph. Scatter plots are created using the `scatter()` function in Matplotlib.

```
# Example data
x = np.random.rand(100)
y = np.random.rand(100)

# Creating a scatter plot
plt.scatter(x, y, color='b', alpha=0.5)
plt.title("Random Scatter Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

Output:



The code displayed the scatter plot of random points with blue markers and some transparency. The appearance of the plot is determined by the following parameters in the code:

- `color='b'`: This sets the color of the scatter plot markers to blue.
- `alpha=0.5`: This sets the transparency level of the markers to 50%.

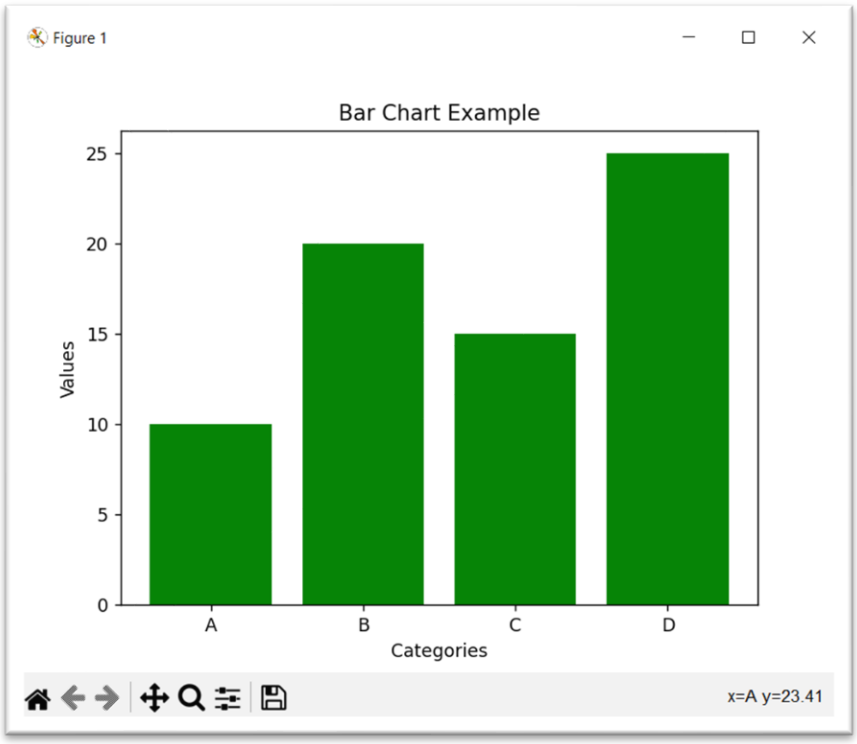
These parameters in the `plt.scatter()` function customize the visual style of the scatter plot, making the distribution of points more distinguishable. The blue color ensures the plot stands out, and the transparency allows for better visualization of overlapping points, providing a clearer view of data density. These stylistic choices help making the plot both informative and visually engaging.

Bar Charts

Bar charts are used to represent categorical data with rectangular bars. They can be created using the `bar()` function in Matplotlib.

```
# Example data
categories = ['A', 'B', 'C', 'D']
values = [10, 20, 15, 25]

# Creating a bar chart
plt.bar(categories, values, color='g')
plt.title("Bar Chart Example")
plt.xlabel("Categories")
plt.ylabel("Values")
plt.show()
```

Output:



The code displayed a bar chart with green bars representing the values of each category.

The appearance of the plot is determined by the following parameters in the code:

- `color='g'`: This sets the color of the bars to green.

- `plt.title("Bar Chart Example")`: This adds a title to the chart.
- `plt.xlabel("Categories")`: This labels the x-axis with "Categories".
- `plt.ylabel("Values")`: This labels the y-axis with "Values".

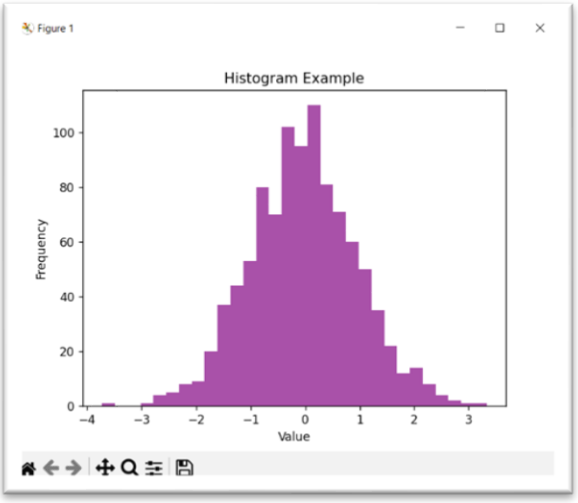
These parameters in the `plt.bar()` function and other `plt` functions customize the visual style of the bar chart, making the data easy to interpret. The green color ensures the bars stand out, and the titles and labels provide context for the data being presented. These stylistic choices help make the plot both informative and visually engaging.

Histograms

Histograms are used to represent the distribution of a dataset. They provide a visual interpretation of numerical data by indicating the number of data points that fall within a range of values (bins). They can be created using the `hist()` function in Matplotlib.

```
# Example data
data = np.random.randn(1000)

# Creating a histogram
plt.hist(data, bins=30, color='purple', alpha=0.7)
plt.title("Histogram Example")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

output:



The code displays a histogram of the dataset with 30 bins and purple bars with some transparency. The appearance of the plot is determined by the following parameters in the code:

- `bins=30`: This sets the number of bins (intervals) to 30, which affects the granularity of the distribution representation.
- `color='purple'`: This sets the color of the bars to purple.
- alpha=0.7: This sets the transparency level of the bars, with 1 being fully opaque and 0 being fully transparent.
- `plt.title("Histogram Example")`: This adds a title to the histogram.
- `plt.xlabel("Value")`: This labels the x-axis with "Value", indicating the variable being measured.
- `plt.ylabel("Frequency")`: This labels the y-axis with "Frequency", indicating the count of data points within each bin.

These parameters in the `plt.hist()` function and other `plt` functions customize the visual style of the histogram, making the distribution of the data easy to interpret. The purple color and transparency level make the bars visually distinct, while the titles and labels provide context for the data being presented. These stylistic choices help make the plot both informative and visually engaging.

3. Advanced Plot Customizations

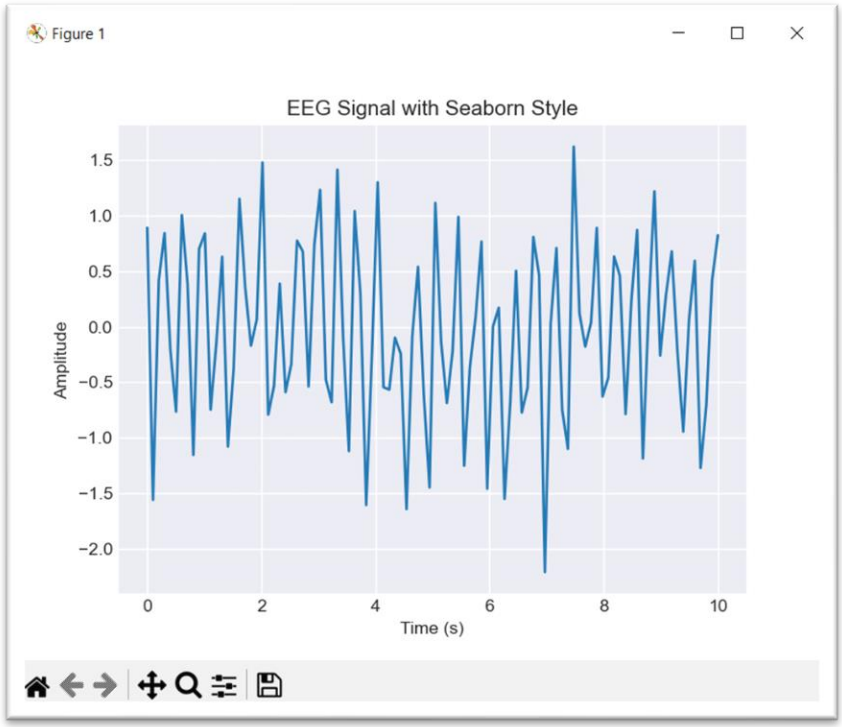
Enhancing Plot Aesthetics

Using Styles and Themes

Matplotlib provides various styles and themes to enhance the aesthetics of your plots. You can easily apply these styles to give your plots a professional look.

```
# Example data
time = np.linspace(0, 10, 100)
eeg_signal = np.sin(2 * np.pi * 7 * time) + 0.5 * np.random.randn(100)

# Applying a style
plt.style.use('seaborn-darkgrid')

# Creating a plot with the selected style
plt.plot(time, eeg_signal)
plt.title("EEG Signal with Seaborn Style")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.show()
```

Output:



The code displays an EEG signal plot using the 'seaborn-darkgrid' style, which enhances the plot with a more polished appearance. The appearance of the plot is determined by the following parameters in the code:

- `plt.style.use('seaborn-darkgrid')`: This applies the 'seaborn-darkgrid' style to the plot, giving it a clean and professional look with a dark grid background.
- `plt.plot(time, eeg_signal)`: This creates the line plot of the EEG signal against time.
- `plt.title("EEG Signal with Seaborn Style")`: This adds a title to the plot.
- `plt.xlabel("Time (s)")`: This labels the x-axis with "Time (s)", indicating the time in seconds.
- `plt.ylabel("Amplitude")`: This labels the y-axis with "Amplitude", indicating the amplitude of the EEG signal.

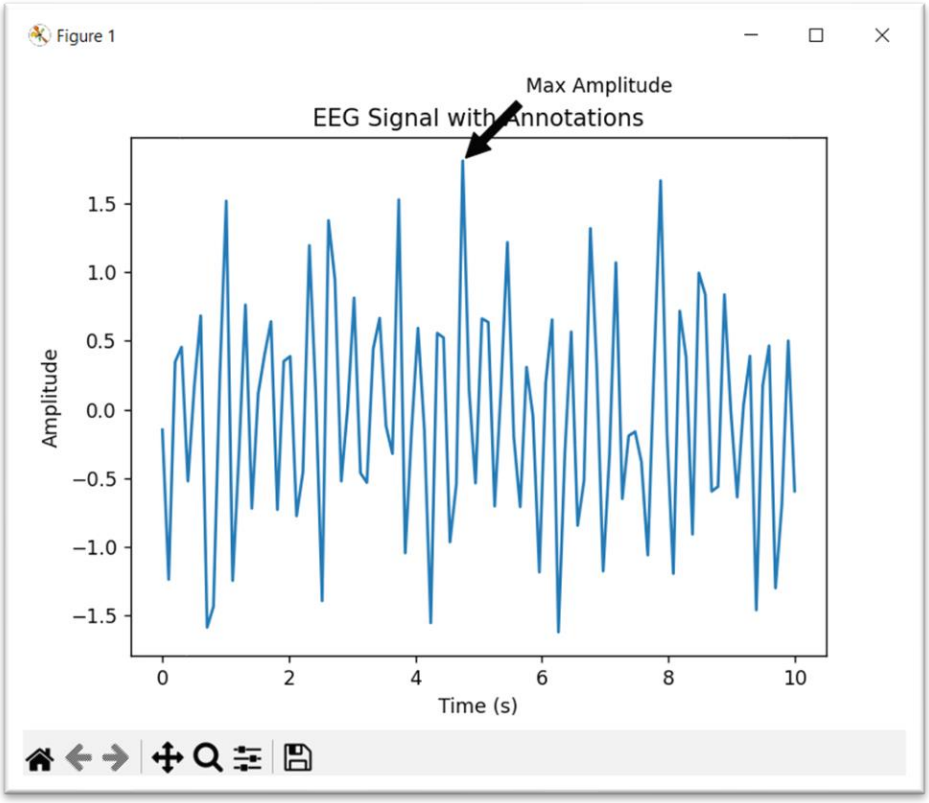
These parameters in the `plt.plot()` function and other `plt` functions customize the visual style of the plot, making the EEG signal easy to interpret. The seaborn-darkgrid style provides a visually appealing background, while the titles and labels give context to the data being presented. These stylistic choices help make the plot both informative and visually engaging.

Adding Annotations and Text

Annotations and text can provide additional context and information on your plots, making them more informative.

```
# Example data
time = np.linspace(0, 10, 100)
eeg_signal = np.sin(2 * np.pi * 7 * time) + 0.5 * np.random.randn(100)

# Creating a plot
plt.plot(time, eeg_signal)
plt.title("EEG Signal with Annotations")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")

# Adding an annotation
max_point = np.argmax(eeg_signal)
plt.annotate('Max Amplitude', xy=(time[max_point], eeg_signal[max_point]),
             xytext=(time[max_point] + 1, eeg_signal[max_point] + 0.5),
             arrowprops=dict(facecolor='black', shrink=0.05))

plt.show()
```

Output:



The code displays an EEG signal plot with an annotation pointing to the maximum amplitude. Here are the new syntactic elements introduced:

- `np.argmax(eeg_signal)`: This function returns the index of the maximum value in the `eeg_signal` array.
- `plt.annotate('Max Amplitude', ...)`: This function adds an annotation to the plot. The parameters include:
  - `xy=(time[max_point], eeg_signal[max_point])`: This sets the location of the annotation arrow to the point of maximum amplitude.
  - `xytext=(time[max_point] + 1, eeg_signal[max_point] + 0.5)`: This sets the location of the annotation text, offset from the arrow point.

- `arrowprops=dict(facecolor='black', shrink=0.05)`: This defines the properties of the annotation arrow, such as its color (`facecolor='black'`) and the shrinking factor of the arrow (`shrink=0.05`).

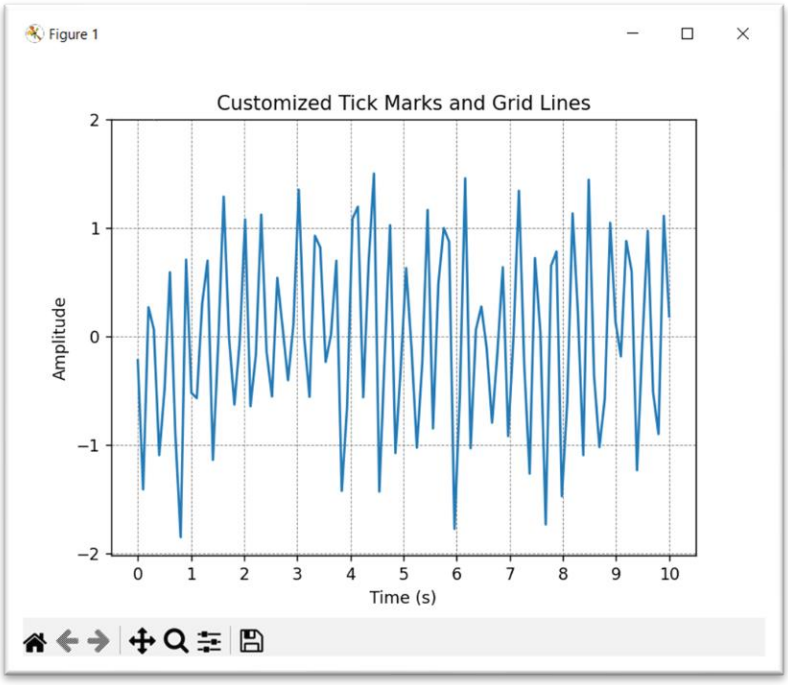
These parameters in the `plt.annotate()` function help to highlight specific points on the plot, adding valuable context and making the plot more informative.

Customizing Tick Marks and Grid Lines

Customizing tick marks and grid lines can make your plots easier to read and interpret.

```
# Example data
time = np.linspace(0, 10, 100)
eeg_signal = np.sin(2 * np.pi * 7 * time) + 0.5 * np.random.randn(100)

# Creating a plot
plt.plot(time, eeg_signal)
plt.title("Customized Tick Marks and Grid Lines")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")

# Customizing tick marks
plt.xticks(np.arange(0, 11, 1))
plt.yticks(np.arange(-2, 3, 1))

# Customizing grid lines
plt.grid(color='grey', linestyle='--', linewidth=0.5)

plt.show()
```

Output:



The code displays an EEG signal plot with customized tick marks and grid lines. The new syntactic elements introduced are:

- `plt.xticks(np.arange(0, 11, 1))`: This function sets the x-axis tick marks at intervals of 1 from 0 to 10.
- `plt.yticks(np.arange(-2, 3, 1))`: This function sets the y-axis tick marks at intervals of 1 from -2 to 2.
- `plt.grid(color='grey', linestyle='--', linewidth=0.5)`: This function customizes the grid lines with the following parameters:
    - `color='grey'`: This sets the color of the grid lines to grey.
    - `linestyle='--'`: This sets the style of the grid lines to dashed.
    - `linewidth=0.5`: This sets the width of the grid lines to 0.5.

These customizations improve the readability of the plot by providing clear and distinct tick marks and grid lines, making it easier to interpret the data.

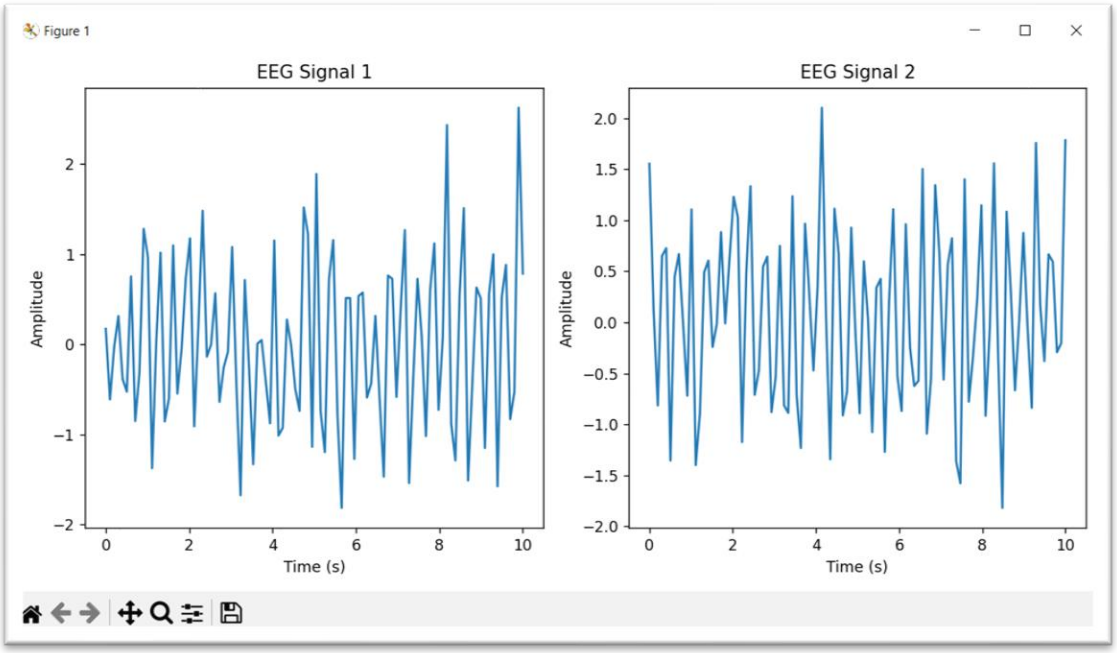
Subplots and Layouts

Creating Multiple Plots in a Single Figure with subplot()

Using `subplot()`, you can create multiple plots within a single figure, which is useful for comparing different datasets.

```
# Example data
time = np.linspace(0, 10, 100)
eeg_signal1 = np.sin(2 * np.pi * 7 * time) + 0.5 * np.random.randn(100)
eeg_signal2 = np.cos(2 * np.pi * 7 * time) + 0.5 * np.random.randn(100)

# Creating subplots
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.plot(time, eeg_signal1)
plt.title("EEG Signal 1")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")

plt.subplot(1, 2, 2)
plt.plot(time, eeg_signal2)
plt.title("EEG Signal 2")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")

plt.tight_layout()
plt.show()
```

Output:



The code displays two EEG signal plots side by side in a single figure. The new syntactic elements introduced are:

- `plt.figure(figsize=(10, 5))`: This function initializes a new figure with a specified size. The `figsize` parameter takes a tuple `(width, height)` in inches, setting the size of the entire figure. In this example, the figure is 10 inches wide and 5 inches tall.
- `plt.subplot(1, 2, 1)`: This function creates a subplot within the figure. The parameters (1, 2, 1) mean that the subplot grid has 1 row and 2 columns, and this subplot is the first one (i.e., the left plot). Similarly, `plt.subplot(1, 2, 2)` creates the second subplot in the grid (i.e., the right plot).
- `plt.tight_layout()`: This function adjusts the subplot parameters to give specified padding. This ensures that the subplots fit within the figure area without overlapping. It automatically adjusts the spacing between the subplots for a neat layout.

These functions work together as follows:

1. `plt.figure(figsize=(10, 5))` creates a large figure to hold multiple subplots.

2. `plt.subplot(1, 2, 1)` and `plt.subplot(1, 2, 2)` divide the figure into a grid of subplots (1 row by 2 columns) and specify the position of each subplot within the grid.
3. `plt.tight_layout()` ensures that the subplots are laid out cleanly without overlapping, adjusting the spacing between them for better visual clarity.

Together, these commands allow you to create and neatly arrange multiple plots within a single figure, making it easier to compare different datasets side by side.

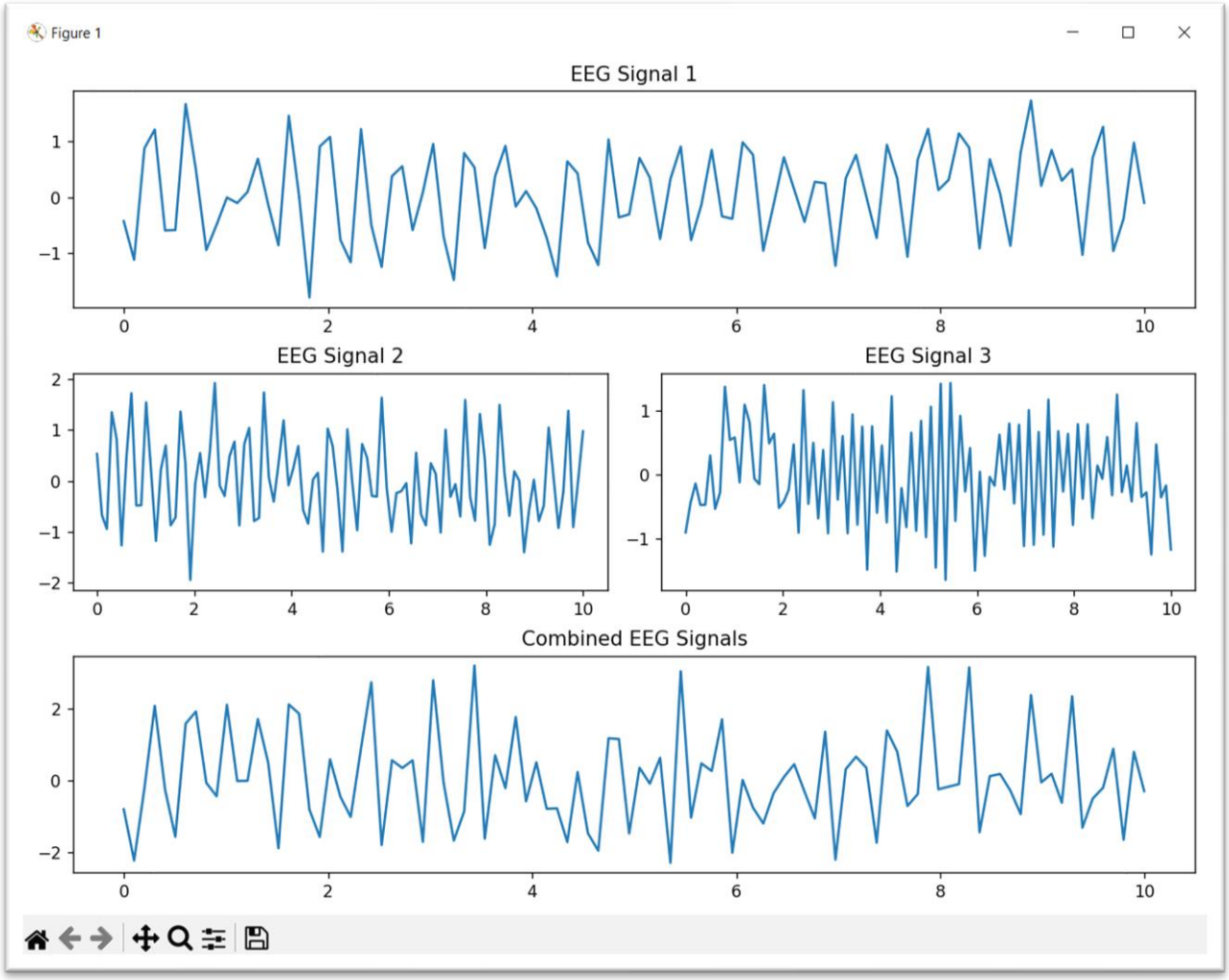
Using GridSpec for More Complex Layouts

`GridSpec` allows for more complex and flexible layouts than `subplot()`. Remember to install `GridSpec` using `pip` and import it.

```
# Importing GridSpec
import matplotlib.gridspec as gridspec

# Example data (Creating 3 imaginary EEG signals)
time = np.linspace(0, 10, 100)
eeg_signal1 = np.sin(2 * np.pi * 7 * time) + 0.5 * np.random.randn(100)
eeg_signal2 = np.cos(2 * np.pi * 7 * time) + 0.5 * np.random.randn(100)
eeg_signal3 = np.sin(2 * np.pi * 5 * time) + 0.5 * np.random.randn(100)

# Creating a complex layout with GridSpec
fig = plt.figure(figsize=(10, 8))
gs = gridspec.GridSpec(3, 2)

ax1 = fig.add_subplot(gs[0, :])
ax1.plot(time, eeg_signal1)
ax1.set_title("EEG Signal 1")

ax2 = fig.add_subplot(gs[1, 0])
ax2.plot(time, eeg_signal2)
ax2.set_title("EEG Signal 2")

ax3 = fig.add_subplot(gs[1, 1])
ax3.plot(time, eeg_signal3)
ax3.set_title("EEG Signal 3")
```

```
ax4 = fig.add_subplot(gs[2, :])
ax4.plot(time, eeg_signal1 + eeg_signal2 + eeg_signal3)
ax4.set_title("Combined EEG Signals")

plt.tight_layout()
plt.show()
```

Output:



The code displays a figure with a complex layout consisting of multiple EEG signal plots.

The new syntactic elements introduced are:

- `import matplotlib.gridspec as gridspec`: This imports the GridSpec module from Matplotlib, which allows for creating more complex subplot layouts.
- `fig = plt.figure(figsize=(10, 8))`: This initializes a new figure with a specified size of 10 inches by 8 inches.
- `gs = gridspec.GridSpec(3, 2)`: This creates a GridSpec object with 3 rows and 2 columns, defining a grid layout for the subplots.
- `fig.add_subplot(gs[0, :])`: This adds a subplot that spans the entire first row. The `gs[0, :]` notation specifies the first row and all columns.
- `fig.add_subplot(gs[1, 0])`: This adds a subplot in the second row and first column. The gs[1, 0] notation specifies the second row and first column.
- `fig.add_subplot(gs[1, 1])`: This adds a subplot in the second row and second column. The gs[1, 1] notation specifies the second row and second column.
- `fig.add_subplot(gs[2, :])`: This adds a subplot that spans the entire third row. The gs[2, :] notation specifies the third row and all columns.
- `plt.tight_layout()`: This function adjusts the subplot parameters to give specified padding, ensuring that the subplots fit within the figure area without overlapping.

These functions work together as follows:

1. `fig = plt.figure(figsize=(10, 8))` creates a large figure to hold multiple subplots.
2. `gs = gridspec.GridSpec(3, 2)` defines a grid layout with 3 rows and 2 columns.
3. `fig.add_subplot()` adds subplots to the specified grid positions, allowing for more complex and flexible layouts compared to the basic subplot() function.
4. `plt.tight_layout()` ensures that the subplots are laid out cleanly without overlapping, adjusting the spacing between them for better visual clarity.

Together, these commands allow you to create and neatly arrange multiple plots within a single figure, making it easier to compare different datasets and display complex layouts.

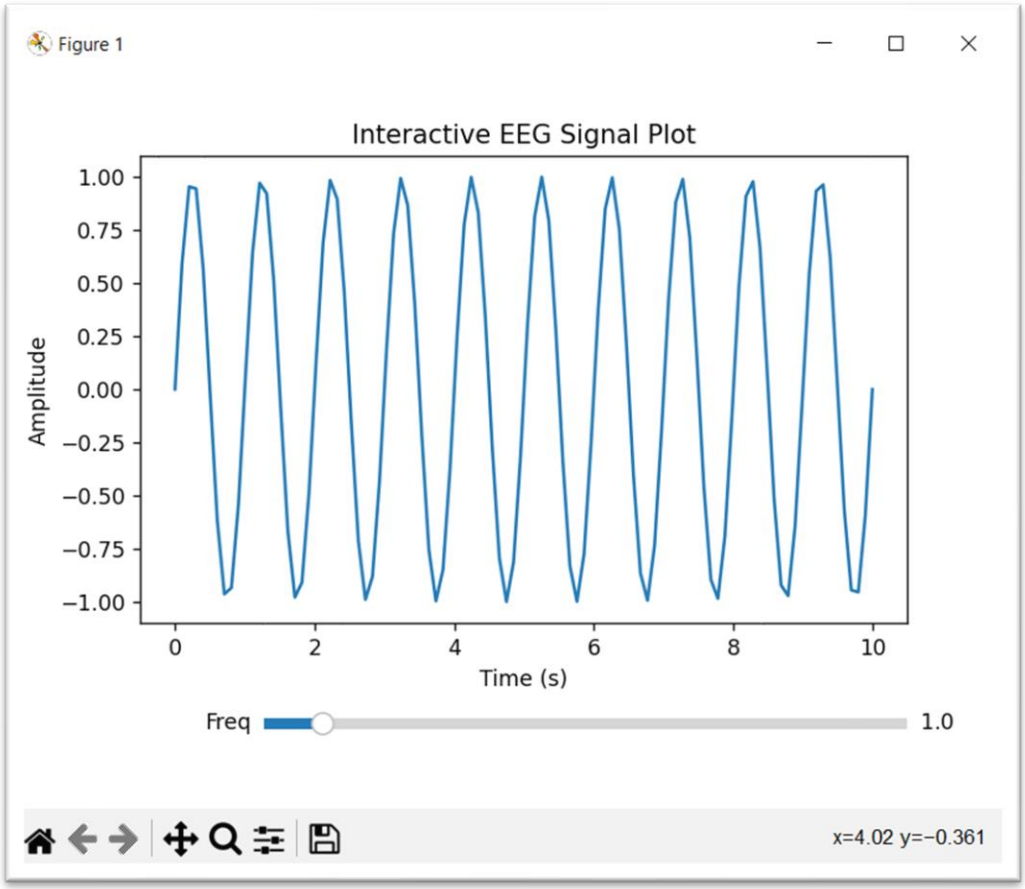
Interactive Plots

Creating Interactive Plots with Matplotlib Widgets

Matplotlib provides widgets for creating interactive plots, such as sliders and buttons.

```
from matplotlib.widgets import Slider

# Example data
time = np.linspace(0, 10, 100)
freq = 1
eeg_signal = np.sin(2 * np.pi * freq * time)

# Creating a plot
fig, ax = plt.subplots()
plt.subplots_adjust(bottom=0.25)
l, = plt.plot(time, eeg_signal)
plt.title("Interactive EEG Signal Plot")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")

# Creating a slider
axfreq = plt.axes([0.25, 0.1, 0.65, 0.03])
sfreq = Slider(axfreq, 'Freq', 0.1, 10.0, valinit=freq)

# Updating the plot based on the slider value
def update(val):
    freq = sfreq.val
    l.set_ydata(np.sin(2 * np.pi * freq * time))
    fig.canvas.draw_idle()

sfreq.on_changed(update)

plt.show()
```

Output:



The code displays an interactive EEG signal plot with a slider that allows users to adjust the frequency of the signal. The new syntactic elements introduced are:

- `from matplotlib.widgets import Slider`: This imports the Slider widget from Matplotlib, which allows for creating interactive sliders in the plot.
- `plt.subplots_adjust(bottom=0.25)`: This function adjusts the space at the bottom of the plot to make room for the slider.
- `axfreq = plt.axes([0.25, 0.1, 0.65, 0.03])`: This function creates an axes object for the slider. The parameters define the position and size of the slider ([left, bottom, width, height]).

- `sfreq = Slider(axfreq, 'Freq', 0.1, 10.0, valinit=freq)`: This creates a slider object. The parameters are:
  - `axfreq`: The axes object where the slider is placed.
  - `'Freq'`: The label for the slider.
  - `0.1, 10.0`: The minimum and maximum values for the slider.
  - `valinit=freq`: The initial value of the slider.
- `def update(val)`: This function defines the behavior when the slider value changes. It updates the y-data of the plot based on the slider's current value.
- `sfreq.on_changed(update)`: This connects the slider to the update function, so that any change in the slider's value triggers the update function.

What Makes the Uploaded Figure Interactive?

The uploaded figure is interactive due to the inclusion of a slider widget that allows the user to dynamically change the frequency of the EEG signal displayed in the plot. Here are the key interactive elements:

- **Slider Widget**: The slider labeled 'Freq' lets the user adjust the frequency of the EEG signal from 0.1 to 10.0. This slider is positioned below the plot.
- **Dynamic Plot Update**: As the user moves the slider, the update function is called, which recalculates the EEG signal with the new frequency value and updates the plot in real-time. This immediate visual feedback makes the plot interactive.

These interactive features enable the user to explore how changing the frequency affects the EEG signal, enhancing the plot's functionality and making the data analysis process more engaging and informative.