Imports:

import scipy.signal

from ..misc import NeurokitWarning, as\_vector

from ..signal import signal\_filter, signal\_smooth

* **scipy.signal**: A library from SciPy that provides powerful signal processing tools, like filters.
* **NeurokitWarning**: A custom warning type used within the Neurokit library.
* **as\_vector**: A helper function to make sure the input signal is in a simple, consistent list-like format.
* **signal\_filter, signal\_smooth**: Other functions from within Neurokit that perform filtering and smoothing operations.

eda\_signal = as\_vector(eda\_signal)

This ensures that your input eda\_signal is converted into a standard, one-dimensional array format, which is easier for the subsequent functions to handle.

Handling Missing Values:

if n\_missing > 0:

warn("There are " + str(n\_missing) + " missing values. "

"Filling missing values...",

category=NeurokitWarning)

eda\_signal = \_eda\_clean\_missing(eda\_signal)

* **if n\_missing > 0:**: If there is at least one missing data point.
* **warn(...)**: A warning message is printed to the console to let you know that data was missing and is being filled.
* **eda\_signal = \_eda\_clean\_missing(eda\_signal)**: The function calls a helper function (\_eda\_clean\_missing) to fill in the gaps. The original signal is then replaced by this repaired version.

n\_missing = np.sum(np.isnan(eda\_signal))

This line checks for any missing values in your signal.

* **np.isnan(eda\_signal)**: Creates a new array where True marks a missing value (Not a Number or NaN) and False marks a valid number.
* **np.sum(...)**: Counts the number of True values, effectively counting all the missing points.

method = method.lower() # Remove capitalized letters

if method == "biosppy":

clean = \_eda\_clean\_biosppy(eda\_signal, sampling\_rate)

elif method in ["default", "neurokit", "nk"]:

clean = \_eda\_clean\_neurokit(eda\_signal, sampling\_rate)

elif method is None or method == "none":

clean = eda\_signal

else:

raise ValueError("NeuroKit error: eda\_clean(): 'method'

should be one of 'biosppy'.")

* **method = method.lower()**: Converts the method name to lowercase so that "BioSPPy" and "biosppy" are treated the same.
* **if method == "biosppy":**: If you choose the "biosppy" method, it calls the \_eda\_clean\_biosppy helper function.
* elif method in ["default", "neurokit", "nk"]:: If you choose "neurokit" (or one of its aliases), it calls the \_eda\_clean\_neurokit helper function.
* elif method is None or method == "none":: If you choose "none", it does nothing and just returns the signal as is.
* else:: If you provide a method name that doesn't exist, it raises an error to stop the program and inform you of the mistake.

return clean

Finally, the function returns the processed, cleaned signal.

**The Helper Functions**

These functions (starting with an underscore \_) do the actual work.

\_eda\_clean\_missing(eda\_signal)

def \_eda\_clean\_missing(eda\_signal):

eda\_signal = pd.DataFrame.pad(pd.Series(eda\_signal))

return eda\_signal

* **pd.Series(eda\_signal)**: Converts the signal into a pandas Series object, which has powerful tools for handling missing data.
* **pd.DataFrame.pad(...)**: This is an older way to do what is now called "forward fill" (ffill). It takes the last valid number and carries it forward to fill any subsequent gaps.

**Example:**

If a signal is [10, 11, NaN, NaN, 14], forward filling turns it into:  
-> [10, 11, 11, 11, 14]

It is like filling a pothole with the same material as the road leading up to it.

def \_eda\_clean\_neurokit(eda\_signal, sampling\_rate=1000):

if sampling\_rate <= 6:

warn(

"EDA signal is sampled at very low frequency. "

"Skipping filtering.",

category=NeurokitWarning,

)

return eda\_signal

# Filtering

filtered = signal\_filter(

eda\_signal,

sampling\_rate=sampling\_rate,

highcut=3,

method="butterworth",

order=4,

)

return filtered

This is the default cleaning method for NeuroKit.

* **if sampling\_rate <= 6:**: It first checks if your sampling rate is very low. You can't filter out high-frequency noise if your signal was recorded too slowly to even capture that noise in the first place. So, if the rate is low, it skips filtering and returns the signal as is.
* signal\_filter(…): This is the main step. It applies a low-pass filter. Think of this like sifting flour. You want to keep the fine flour (the real EDA signal) and get rid of the big lumps (the high-frequency noise).
* highcut=3: This sets the cutoff. Any signal component that changes faster than 3 times per second (3Hz) is considered “noise” and is removed.
* method=”butterworth”, order=4 : This specifies the type of filter. A Butterworth filter is a standard, effective choice, and an order of 4 makes it reasonably steep, meaning it does it does a good job of separating the signal from the noise.