```
import warnings
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from scipy.io import loadmat
import scipy.signal as sig
import simpleaudio

def get_windows(data, width, overlap, copy=True):
    """
    See here: https://stackoverflow.com/a/45730836, adapted to my
    preferred args
    """
    step = width - overlap
    sh = (data.size - width + 1, width)
    st = data.strides * 2
    view = np.lib.stride_tricks.as_strided(data.T, strides=st, shape=sh, writeable=False)[0::s
tep]
    if copy:
        return view.copy()
    else:
        return view

def apply_window_function(windowed_data, windowing_function):
    # Not pythonic to LBYL but this could be slow so thbbbtttt
    if windowing_function.ndim != 1:
        raise ValueError("Windowing function must be 1 dimensional!")
    if windowed_data.ndim != 2:
        raise ValueError("Windowed data must be 2D ndarray!")
    if windowed_data.shape[1] != windowing_function.shape[0]:
        print("windowed_data.shape: {}, windowing_function.shape: {}".format(windowed_data.sha
pe, windowing_function.shape))
        raise ValueError("Windowing function length != data window length!")

    return windowed_data*windowing_function


def spectrogram(data, fs=1.0, window=None, nperseg=None, noverlap=None):
    """
    Return the calculated spectrogram in a fashion similar to SciPy’s API.
    Assumes input data is real, therefore the FFT is symettric, thus returning only
    the frequencies >=0.
    """
    if data.ndim != 1:
        raise ValueError("We only support 1D data.")
    if nperseg is None:
        nperseg = 256
    if nperseg > data.size:
        warnings.warn("nperseg larger than data, reducing...")
        nperseg = data.size
    if noverlap is None:
        noverlap = np.int(np.floor(nperseg / 8))

    if window is None:
        # Shannon/Boxcar/Square/Dirichlet Window (or ’no’ window)
        window = np.ones(nperseg)

    if window.ndim != 1:
        raise ValueError("Window must be 1D!")
    if window.size != nperseg:
        print("nperseg: {}, window.size: {}".format(nperseg, window.size))
        raise ValueError("Windowing function must be the size of the window!")
```

```
    chunked_data = get_windows(data, nperseg, noverlap)
    windowed_data = apply_window_function(chunked_data, window)

    # Calculate the time and frequency vectors
    t = np.arange(nperseg/2, data.shape[-1] - nperseg/2+1, nperseg-noverlap)/float(fs)
    freqs = (np.arange(1, nperseg + 1, dtype=int) // 2) / float(nperseg*(1.0/fs))

    # the above produces 2 of every frequency but the 0 and max frequency, so take the uniques
    # Slicing proved annoying to get consistent. probably not the most performant but WHATEVAH
    freqs = np.unique(freqs)
    Sxx = np.abs(np.apply_along_axis(np.fft.rfft, 1, windowed_data).T)

    return freqs, t, Sxx


def plot_spectrogram(data, fs=8192, window=None, nperseg=None, noverlap=None, maxfreq=None, no
rmed=None, title=None):
    # Note that maxfreq is inclusive!
    f, t, Sxx = spectrogram(data, fs=fs, window=window, nperseg=nperseg, noverlap=noverlap)

    # normalize the Spectrum
    if normed == "norm":
        normalize = matplotlib.colors.Normalize(vmin=Sxx.min(), vmax=Sxx.max())
    elif normed == "log":
        normalize = matplotlib.colors.LogNorm(vmin=Sxx.min(), vmax=Sxx.max())
    elif normed is not None:
        warning.warn("Normed must be 'log' or 'norm'.")
        normalize = None
    else:
        normalize = None

    if maxfreq is not None:
        indices = np.where(f <= maxfreq)[0]
        f=f[indices]
        if f.size <= 0:
            raise ValueError("maxfreq was too low for the data set provided.")
        Sxx = Sxx[indices][:]

    plt.figure()
    if title:
        plt.title(title, wrap=True)

    plt.pcolormesh(t, f, Sxx, norm=normalize)
    plt.ylabel("Frequency (Hz)")
    plt.xlabel("Time (Seconds)")
    plt.show()

def generate_shannon_window(nperseg):
    return np.ones(nperseg)

def generate_gauss_window(nperseg, sigma):
    # normalize sigma somewhat
    sigma = sigma * (nperseg/10)
    return sig.gaussian(nperseg, sigma)

def generate_sombrero(nperseg, a):
    a = a * nperseg/10
    window = sig.ricker(nperseg, a)
    return window / max(window)

def generate_illuminati(nperseg):
    return sig.triang(nperseg)
```