for start we define some function and initializing some variables and arrays witch we need in future:

notes\_base: This line creates an array of base frequencies for different musical notes. It uses NumPy's arange function to generate an array of numbers from 0 to 11, which are divided by 12 and raised to the power of 2. These values are then multiplied by 27.5 to obtain the frequencies for each note.

notes\_duration: This line creates an array representing the duration of each note in milliseconds. It uses NumPy's array function to create an array with specific values, which are then multiplied by 0.7.

notes\_ann: This line creates a list of note names. Each element corresponds to a note in the notes\_base array.

sin\_wave function: This function generates a sinusoidal waveform. It takes three parameters: the frequency of the waveform (f), the number of samples (n), and the sample rate (fs). It first creates a linearly spaced array x from 0 to 2π, representing the time axis. It then creates another linearly spaced array xp from 0 to -1\*(n\*ring/fs), where ring is set to 30. The function generates a sinusoidal waveform y by taking the sine of the product of x, f, and n/fs, and multiplies it by the exponential of xp. Finally, it creates a 2-dimensional array z with shape (n, 2) and assigns y to both columns of z. The function returns z.

play\_note function: This function plays a single note. It takes four parameters: the note ID (note\_id), the octave (octave), the duration (dur), and the sample rate (fs). If the note\_id is less than 3, it increments the octave by 1. It then calls the sin\_wave function to generate the waveform y for the specified note and duration. It uses sd.play from the sounddevice library to play the waveform y at the specified sample rate. Finally, it uses sd.wait() to wait until the playback is finished and returns.

put\_note function: This function is similar to play\_note, but instead of playing the note, it generates and returns the waveform y for the specified note, octave, duration, and sample rate.

get\_music function: This function takes a list of music notes (music\_notes) and the sample rate (fs). It iterates over each item in music\_notes and calls the put\_note function to generate the waveform y for each note. The generated waveforms are concatenated using NumPy's concatenate function and returned as a single waveform m.

fs1: This variable stores the sample rate of the audio, set to 44100.

music: This list represents a musical composition. Each item in the list corresponds to a note and contains three elements: the note ID, the octave, and the duration.

y: This line calls the get\_music function with the music list and the sample rate fs1. It generates the waveform y for the musical composition.

sd.play(y, fs1): This line plays the waveform y using the sounddevice library at the sample rate fs1.

Overall, this code defines functions to generate and play sinusoidal waveforms for different musical notes and durations. It then composes a piece of music using the defined functions and plays it using the sounddevice library.

\begin{lstlisting}[style=mystyle]

# This is a Python comment

print("Hello, world!")

\end{lstlisting}

\lstdefinestyle{mystyle}{

language=Python,

basicstyle=\ttfamily,

keywordstyle=\color{blue},

commentstyle=\color{gray},

stringstyle=\color{green},

showstringspaces=false,

numbers=left,

numberstyle=\tiny,

breaklines=true,

breakatwhitespace=true,

frame=single,

captionpos=b,

caption={Example Python code},

}

This code utilizes the OpenCV library (cv2) to perform image processing tasks. Here's a breakdown of the code:

template\_path and background\_path: These variables store the paths to the template image and background image files, respectively.

template = cv2.imread(template\_path): This line reads the template image file using the cv2.imread() function and stores it in the template variable. The imread() function loads the image as a NumPy array, representing the image's pixel values.

background = cv2.imread(background\_path): This line reads the background image file using the cv2.imread() function and stores it in the background variable. Similar to the previous line, the imread() function loads the image as a NumPy array.

template\_gray = cv2.cvtColor(template, cv2.COLOR\_BGR2GRAY): This line converts the template image from the BGR color space to grayscale using the cv2.cvtColor() function. Grayscale images contain only one channel representing the intensity of each pixel, while BGR images have three channels for blue, green, and red intensities. The cv2.COLOR\_BGR2GRAY flag specifies the conversion from BGR to grayscale.

background\_gray = cv2.cvtColor(background, cv2.COLOR\_BGR2GRAY): This line converts the background image from the BGR color space to grayscale, similar to the previous line.

After executing this code, you would have the following variables:

template: A NumPy array representing the loaded template image.

background: A NumPy array representing the loaded background image.

template\_gray: A grayscale version of the template image.

background\_gray: A grayscale version of the background image.

The provided code performs template matching using the cv2.matchTemplate() function and extracts the location of the best match. Here's an explanation of the code:

result = cv2.matchTemplate(background\_gray, template\_gray, cv2.TM\_CCOEFF\_NORMED): This line applies the template matching algorithm to find the best match of the template\_gray image within the background\_gray image. It uses the method cv2.TM\_CCOEFF\_NORMED to compute the normalized cross-correlation coefficient between the template and the background image. The result is stored in the result variable, which is a two-dimensional array representing the match scores at each pixel location.

min\_val, max\_val, min\_loc, max\_loc = cv2.minMaxLoc(result): This line finds the minimum and maximum values, as well as their corresponding locations, in the result array using the cv2.minMaxLoc() function. The minimum and maximum values (min\_val and max\_val) represent the weakest and strongest match scores, respectively. The corresponding locations (min\_loc and max\_loc) indicate the positions where these extreme scores occur.

top\_left = max\_loc: This line assigns the location of the maximum match score to the top\_left variable. Since max\_loc represents the top-left corner of the matched region in the result array, it serves as the starting point of the matched region in the original background image.

bottom\_right = (top\_left[0] + template.shape[1], top\_left[1] + template.shape[0]): This line calculates the coordinates of the bottom-right corner of the matched region in the original background image. It uses the top\_left coordinates along with the width and height of the template image (obtained through template.shape[1] and template.shape[0], respectively) to determine the position of the bottom-right corner.

After executing this code, you would have the following variables:

result: A two-dimensional array representing the match scores at each pixel location.

min\_val: The minimum match score.

max\_val: The maximum match score.

min\_loc: The coordinates of the minimum match score.

max\_loc: The coordinates of the maximum match score.

top\_left: The coordinates of the top-left corner of the matched region in the original background image.

bottom\_right: The coordinates of the bottom-right corner of the matched region in the original background image.

cv2.rectangle(background, top\_left, bottom\_right, (0, 255, 0), 2): This line draws a rectangle on the background image using the cv2.rectangle() function.

By calling this function, a green rectangle is drawn on the background image, highlighting the region that matches the template.