pywebaudioplayer: Bridging the gap between audio processing code in Python and attractive visualisations based on web technology

Johan Pauwels
Centre for Digital Music
Queen Mary University of London
j.pauwels@qmul.ac.uk

Mark B. Sandler
Centre for Digital Music
Queen Mary University of London
mark.sandler@qmul.ac.uk

ABSTRACT

Lately, a number of audio players based on web technology have made it possible for researchers to present their audio-related work in an attractive manner. Tools such as *wavesurfer.js*, *waveform-playlist* and *trackswitch.js* provide highly-configurable players, allowing a more interactive exploration of scientific results that goes beyond simple linear playback.

However, the audio output to be presented is in many cases not generated by the same web technologies. The process of preparing audio data for display therefore requires manual intervention, in order to bridge the resulting gap between programming languages. While this is acceptable for one-time events, such as the preparation of final results, it prevents the usage of such players during the iterative development cycle. Having access to rich audio players already during development would allow researchers to get more instantaneous feedback. The current workflow consists of repeatedly importing audio into a digital audio workstation in order to achieve similar capabilities, a repetitive and time-consuming process.

In order to address these needs, we present *pywebaudioplayer*, a Python package that automates the generation of code snippets for the each of the three aforementioned web audio players. It is aimed at use-cases where audio development in Python is combined with web visualisation. Notable examples are *Jupyter Notebook* and WSGI-compatible web frameworks such as *Flask* or *Django*.

Keywords

audio player, Python, multi-track audio

1. INTRODUCTION

Python is quickly becoming one of the most popular languages for general programming. According to the TIOBE[1] Programming Community Index for April 2018 (figure 1), which attempts to measure programming language popularity, Python is now the fourth most popular language and accounts for 5.803% of all usage with a year-over-year increase of 2.35%. In contrast, JavaScript is in eighth position, with a market share of 3.492% and an increase of 0.64%. Although specific numbers are unavailable, for scientific computing and especially audio processing, the popularity of Python seems even higher [10].

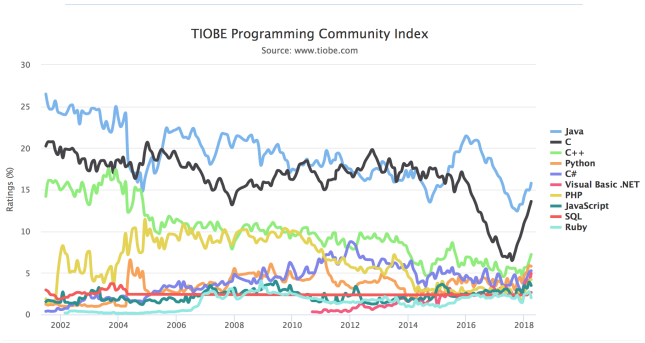

Figure 1: TIOBE Index of programming language popularity (April 2018).

One of the reasons for Python's widespread adoption is the large ecosystem of scientific libraries that surround Python, such as *NumPy* [11] and *SciPy* [6]. For audio processing, libraries such as *essentia* [3], *madmom* [2] or *librosa* [9] provide reusable components to speed up algorithm development. Even in cases when Python is not used for core computation, because its interpreted nature leads to prohibitively slow execution times for instance, it is useful to tie different components together [12] or to plot figures, for example with *matplotlib* [5] or *seaborn* [16].

Another reason for the popularity of Python in scientific research is that it is an interpreted language with an expressive syntax and allows interactive development with the *IPython* [14] shell. In this context, *Jupyter Notebook* [8] is a popular development platform because it allows to mix code with text and multimedia elements in a reproducible way. Since *Jupyter Notebook* is browser-based, web technologies can be used to display multimedia elements. When developing audio processing code, it can be really useful to be able to play back audio within a *Jupyter Notebook* itself. The default audio player is very minimal though. Providing an richer alternative for this default audio player is the main

[1] https://www.tiobe.com/tiobe-index/

Web Audio Conference WAC-2018, September 19–21, 2018, Berlin, Germany.

Figure 2: Comparison between standard audio player and wavesurfer with default options in *Jupyter Notebook*.

motivation for this work.

The envisioned usage goes beyond *Jupyter Notebook* though, but includes all contexts where Python code is used to develop web applications. A good example are WSGI-compatible frameworks such as *Flask* [15] or *Django* [4]. To this end, no *Jupyter* specific functionality is included, but pure HTML and JavaScript is generated as strings by *pywebaudioplayer*. Rendering these strings is then left to the mechanism provided by each specific context. For example, this would mean passing the string to `IPython.display.HTML` when *Jupyter Notebook* is used or to an HTML template in the case of *Flask* and *Django*.

Three popular JavaScript/HTML5 audio players are wrapped by *pywebaudioplayer*, namely *wavesurfer.js* [7], *waveform-playlist* [1] and *trackswitch.js* [17]. They have been selected because of the distinct use-cases they excel at. Each of them is presented together with a scenario for which it is most useful in sections §2, §3 and §4 respectively. We end this paper with some conclusions and a look at future work in section §5.

Technically speaking, the wrapper is rather straightforward. Three Python functions corresponding to each of the three audio players have been defined in a single package, which return HTML code snippets as strings. These snippets are constructed in Python based on the given arguments. The underlying Javascript code is kept unmodified in all cases. For now, these libraries are loaded from the web, but work is under way to allow them to be loaded from local copies bundled into the installed Python package as well, such that offline usage is possible.

We aimed to define a programming interface that tries to maximise consistency between the three players by making the function signatures similar. Comparable functionality has been assigned the same parameter names, in order to facilitate changing between players, while each player's default parameter values have been kept.

2. WAVESURFER: SINGLE AUDIO FILE

In the simplest research scenario, one is working on transforming or generating a single audio file and simple playback for aural verification is all that is required. Nonetheless, in this case a waveform display helps to get a quick visual overview of the audio and advanced controls such as forward and backward buttons help to navigate around the track. As evident from the comparison in figure 2, *wavesurfer* is able to provide these features that the basic HTML5 audio player in *Jupyter Notebook* is lacking.

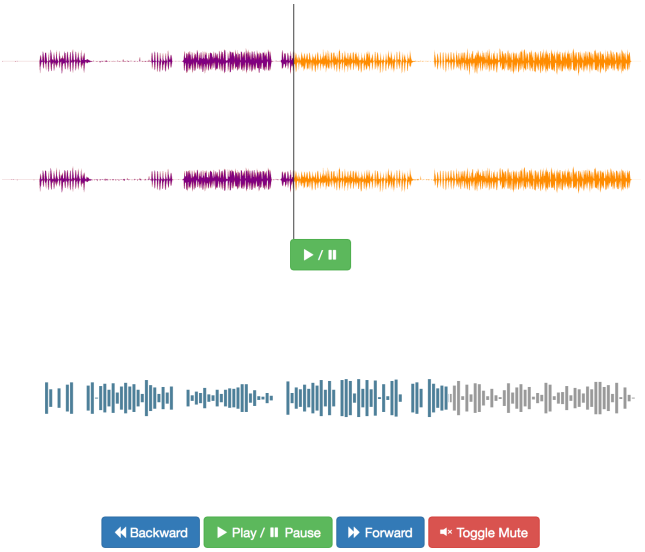

Figure 3: Output of listing 1 rendered by the *Flask* web framework.

Listing 1 demonstrates how *pywebaudioplayer* works. Configuration options for the different players are organised into three categories: *controls*, *display* and *behaviour*. Each of these three is determined by setting values in a Python dictionary, where absence of a particular option name means that the default value is being used. Not passing a dictionary at all means that all values of that category will be set to their defaults. Together with the path to an audio file, the dictionaries determine the player configuration.

Listing 1: Wavesurfer example

```
import pywebaudioplayer as pwa
controls1 = {
  'text_controls': False,
  'backward_button': False,
  'forward_button': False,
  'mute_button': False}
display1 = {
  'unplayed_wave_colour': 'darkorange',
  'played_wave_colour': 'purple',
  'height': 128}
behaviour1 = {'mono': False}
w1 = pwa.wavesurfer(path, controls1,
  display1, behaviour1)

controls2 = {
  'text_controls': True,
  'backward_button': True,
  'forward_button': True,
  'mute_button': True}
display2 = {
  'unplayed_wave_colour': '#999',
  'played_wave_colour':
    'hsla(200, 100%, 30%, 0.5)',
  'cursor_colour': '#fff',
  'bar_width': 3,
  'height': 256}
w2 = pwa.wavesurfer(path, controls2, display2)
```

Table 1: Keys in the *controls* dictionary with their default values.

controls	wavesurfer	waveform_playlist	description
text_controls	✓ [True]	✓ [False]	add text labels to buttons
backward_button	✓ [True]	✓ [True]	show backward search button
forward_button	✓ [True]	✓ [True]	show forward search button
stop_button	✗	✓ [True]	show stop button
pause_button	✗	✓ [True]	show pause button
mute_button	✓ [True]	✗	show mute button
track_controls	✗	✓ [True]	show per track control section with gain, mute and solo
track_controls_width	✗	✓ [200]	width in pixels of the per track control section

Table 2: Keys in the *display* dictionary with their default values.

display	wavesurfer	waveform_playlist	description
height	✓ [128]	✓ [100]	height in pixels of individual waveform canvas
background_colour	✗	✓ ['white']	colour of waveform canvas
unplayed_wave_colour	✓ ['#999']	✗	waveform colour before being play
played_wave_colour	✓ ['#555']	✗	waveform colour after being played
cursor_colour	✓ ['#333']	✗	cursor colour

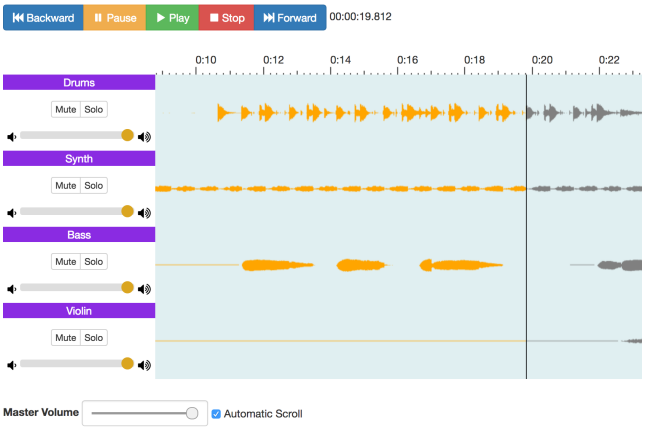

Figure 4: Output of listing 2 rendered by the *Flask* web framework.

Wavesurfer.js does not contain an integrated control section, but leaves it open to the user in order to provide maximum flexibility. To make our Python wrapper self-contained, we added basic control buttons using *bootstrap* [13] based on example code on the *wavesurfer.js* website. Their appearance can be controlled by the keys in the *controls* dictionary described in table 1. Similarly, the display and behaviour of the player can be controlled by the keys in table 2 and table 3, respectively. The output of listing 1 can be seen in figure 3, which demonstrates the extent of possible player customisations.

3. WAVEFORM_PLAYLIST: MULTI-TRACK AUDIO WITH DAW INTERFACE

Some types of research produce multiple audio files at once, which should be previewed in sync. Examples are source separation or auralisation of transcriptions that are best listened to together with the source material. For these cases, the standard HTML5 player nor *wavesurfer* suffice. *Waveform_playlist*[2] provides a user interface similar to a digital audio workstation (DAW) like Logic Pro, ProTools or Cubase. Multiple tracks are presented as stacked waveform displays, each of which can be muted or soloed.

We no longer specify a path to an audio file, but an array of *track* dictionaries as first argument, whose keys can be found in table 4. At minimum, a *path* key needs to be present to specify where the audio needs to be loaded from. For convenience, raw samples can be passed too, which get written first to the given path. The second, third and fourth argument are still the *controls*, *display* and *behaviour* dictionaries of tables 1, 2 and 3, such that function calls are similar to the example in listing 2. The result of this listing can be seen in figure 4.

Listing 2: Waveform_playlist example

```
import pywebaudioplayer as pwa
tracks = [
  {'title': 'Drums', 'path': 'drums.mp3'},
  {'title': 'Synth', 'path': 'synth.mp3'},
  {'title': 'Bass', 'path': 'bass.mp3'},
  {'title': 'Violin', 'path': 'violins.mp3'}]
wp = pwa.waveform_playlist(tracks,
  {'text_controls': True},
  {'background_colour': '#E0EFF1'},
  {'mono': True})
```

[2]Note that the JavaScript library *waveform-playlist* contains a *hyphen* in its name, following JavaScript naming conventions, whereas we named our wrapper *waveform_playlist* with an underscore to follow Python conventions. In this text, both versions will appear depending on whether the original library or the wrapper is being referenced, similar to the difference in usage between *wavesurfer.js* and *wavesurfer*.

Table 3: Keys in the *behaviour* dictionary with their default values.

behaviour	wavesurfer	waveform_playlist	description
mono	✓ [True]	✓ [False]	downmix stereo files to mono
normalise	✓ [False]	✗	normalise to have peak amplitude of 1

Table 4: Keys in the *track* dictionary with their default values.

track	waveform_playlist	trackswitch	description
path	✓ (required)	✓ [None]	path to track audio file
title	✓ [None]	✓ [None]	name of the track
samples	✓[a] [None]	✓ [None]	tuple consisting of list/numpy array with raw samples and samplerate
mimetype	✗	✓ [None]	mime type of the audio file
colour	✗	✓ [None]	individual track colour
solo	✓ [False]	✗	solo button activated at start
mute	✓ [False]	✗	mute button activated at start
gain	✓ [1]	✗	gain
image	✗	✓ [None]	path to track image file

[a] only used in combination with path

```
In [3]:
1 import pywebaudioplayer as pwa
2 IPython.display.HTML(pwa.trackswitch(tracks,
3     text='Example trackswitch.js instance.', seekable_image='mix.png', seek_margin=(4,4)))
```

Out[3]:

Figure 5: Example of *trackswitch* in *Jupyter Notebook*.

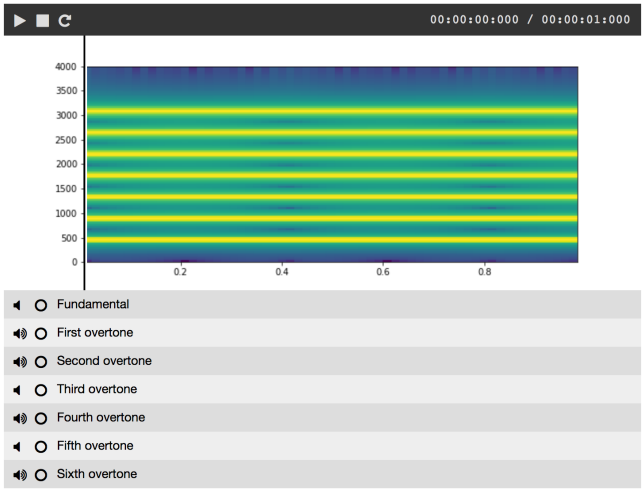

Figure 6: Output of listing 3 in *Jupyter Notebook*.

4. TRACKSWITCH: MULTI-TRACK AUDIO WITH CUSTOM VISUALISATION

The last player that is wrapped by *pywebaudioplayer* is *trackswitch.js*. Like *waveform_playlist*, it is a multi-track player, which means that we again pass a list of *track* dictionaries as the first, obligatory argument to the function. For this player, raw samples can be embedded directly, without the need to specify a *path* to write them to disk, but the latter is still recommended in all but the smallest test cases, since embedding large files in HTML code can make a page unresponsive. Because *trackswitch* has relatively little configuration options compared to the other two players, the *controls*, *display* and *behaviour* dictionaries are not used in this case. All configuration is done by optional function arguments, an overview of these can be found in table 5.

The advantage of *trackswitch* is that it is possible to combine it with custom visualisations. All that is required is to pass any image file along to the player, leaving the way the image is created open for the user to decide. A cursor can be set to scroll over an image in sync with the audio by specifying the active area of the image with the `seekable_image` and `seek_margin` arguments. This functionality allows us to recreate the example of the *trackswitch.js* documentation[3] in figure 5, where the `tracks` array from listing 2 is reused.

However, determining the right offsets to line up the cursor with the appropriate area of the image can be a hassle. Therefore we added the possibility to pass a `matplotlib.figure.Figure` object as `seekable_image` argument, such that the `seek_margin` can be read directly from the figure. In listing 3, we give an example that demonstrates figure generation with *matplotlib* for *trackswitch*. Its output can be seen in figure 6.

Listing 3: Trackswitch example with automatic figure generation

```
import numpy as np
samplerate = 8000
freq = 440
duration = 1
t = np.arange(duration*samplerate)
f0 = np.sin(2*np.pi*freq*t/samplerate)
f1 = np.sin(2*np.pi*2*freq*t/samplerate)
f2 = np.sin(2*np.pi*3*freq*t/samplerate)
f3 = np.sin(2*np.pi*4*freq*t/samplerate)
f4 = np.sin(2*np.pi*5*freq*t/samplerate)
f5 = np.sin(2*np.pi*6*freq*t/samplerate)
f6 = np.sin(2*np.pi*7*freq*t/samplerate)
complex_sine = f0+f1+f2+f3+f4+f5+f6

import matplotlib.pyplot as plt
fig, ax = plt.subplots(ncols=1, figsize=(10,4))
ax.specgram(complex_sine, Fs=samplerate,
            detrend='none')

import pywebaudioplayer as pwa
ts = pwa.trackswitch([
  {'title': 'Fundamental',
   'samples': (f0, samplerate),
   'path': 'f0.wav'},
  {'title': 'First overtone',
   'samples': (f1, samplerate),
   'path': 'f1.wav'},
  {'title': 'Second overtone',
   'samples': (f2, samplerate),
   'path': 'f2.wav'},
  {'title': 'Third overtone',
   'samples': (f3, samplerate),
   'path': 'f3.wav'},
  {'title': 'Fourth overtone',
   'samples': (f4, samplerate),
   'path': 'f4.wav'},
  {'title': 'Fifth overtone',
   'samples': (f5, samplerate),
   'path': 'f5.wav'},
  {'title': 'Sixth overtone',
   'samples': (f6, samplerate),
   'path': 'f6.wav'}],
  seekable_image=(fig, 'spectrogram.png'),
  repeat=True)
```

[3]https://audiolabs.github.io/trackswitch.js/configuration.html

5. CONCLUSIONS AND FUTURE WORK

In this paper, we presented *pywebaudioplayer*, a wrapper around three JavaScript/HTML5 audio players that allows them to be conveniently used from within Python. Automating the bridge between Python and web technologies allows those players to be used to get auditive feedback during iterative development, instead of just using them to present final results. We attempted to bring three players together in a consistent programming interface, in order to ease transitioning between them. Each player has its own use-case, going from displaying single audio files to multi-track audio in a DAW interface or with custom visualisation.

A unifying wrapper such as *pywebaudioplayer* will never be able to provide the same flexibility in configuration as using the separate players directly, since that's not its main objective. We rather strive for improving convenience, at the expense of trading in some configurability. Nonetheless, we still plan on increasing the configuration options that are exposed by the Python wrapper. Particularly for *trackswitch*, we plan to add more convenience functions such that it becomes easier to use custom visualisations. One planned addition is to automatically generate visualisations for each of the separate tracks that are similar to the overall visualisation.

In its current state, *pywebaudioplayer* is ready for public alpha testing. To this end, it has been uploaded to

Table 5: Optional arguments for the *trackswitch* function with their defaults. Partly adapted from *trackswitch.js* documentation.

argument name	default value	description
text	”	accompanying text
text_style	None	CSS style for accompanying text
seekable_image	None	path to seekable image or tuple with matplotlib.figure.Figure and path to save Figure to
seek_margin	None	tuple with start and stop offsets as percentage [0-100] of image to align cursor boundaries
images	None	list with paths to additional images
mute	True	show mute buttons
solo	True	show solo buttons
globalsolo	True	mute all other trackswitch instances when playback starts
repeat	False	initialise player with repeat button enabled
radiosolo	False	allow only one track to be soloed at a time
onlyradiosolo	False	sets both mute to False and radiosolo to True in one argument
spacebar	False	bind the spacebar to play/pause
tabview	False	arrange tracks in a tab view

the *PyPI package index* and can be installed using *pip* as `pip install [--user] pywebaudioplayer`. All code, including the examples, is available under a MIT licence on GitHub[4], where there's also an issue tracker to file bugs. The current version is already fully functional in combination with WSGI web frameworks, but the interactivity inherent to *Jupyter Notebooks* still requires some workarounds. In order to improve the user experience, it probably will be necessary to develop a dedicated Jupyter Notebook plugin in the future.

Acknowledgements

This work has been partly funded by the UK Engineering and Physical Sciences Research Council (EPSRC) grant EP/L019981/1 and by the European Union's Horizon 2020 research and innovation programme under grant agreement N° 688382.

References

[1] Naomi Aro. waveform-playlist. 2015. URL: https://naomiaro.github.io/waveform-playlist/.

[2] Sebastian Böck et al. “Madmom: A new Python audio and music signal processing library”. In: *Proceedings of the 2016 ACM Conference on Multimedia*. ACM. 2016, pp. 1174–1178.

[3] Dmitry Bogdanov et al. “ESSENTIA: an open-source library for sound and music analysis”. In: *Proceedings of the 21st ACM international conference on Multimedia*. ACM. 2013, pp. 855–858.

[4] Django. Django Software Foundation. 2005. URL: https://djangoproject.com.

[5] John D. Hunter. “Matplotlib: A 2D graphics environment.” In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[6] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. 2001. URL: http://www.scipy.org/.

[7] katspaugh. wavesurfer.js. 2013. URL: https://wavesurfer-js.org.

[8] Thomas Kluyver et al. “Jupyter Notebooks-a publishing format for reproducible computational workflows”. In: *Proceedings of the 20th International Conference on Electronic Publishing*. 2016, pp. 87–90. DOI: 10.3233/978-1-61499-649-1-87.

[9] Brian McFee et al. “librosa: Audio and music signal analysis in python”. In: *Proceedings of the 14th Python in science conference*. 2015, pp. 18–25.

[10] K Jarrod Millman and Michael Aivazis. “Python for scientists and engineers”. In: *Computing in Science & Engineering* 13.2 (2011), pp. 9–12. DOI: 10.1109/MCSE.2011.36.

[11] Travis E. Oliphant. *A guide to NumPy*. Trelgol Publishing USA, 2006.

[12] Travis E. Oliphant. “Python for scientific computing”. In: *Computing in Science & Engineering* 9.3 (2007). DOI: 10.1109/MCSE.2007.58.

[13] Mark Otto and Jacob Thornton. Bootstrap. 2011. URL: http://getbootstrap.com/.

[14] Fernando Pérez and Brian E. Granger. “IPython: a system for interactive scientific computing”. In: *Computing in Science & Engineering* 9.3 (2007).

[15] Armin Ronacher et al. Flask: a microframework for Python web development. 2010. URL: http://flask.pocoo.org.

[16] Michael Waskom et al. seaborn: v0.8.1 (September 2017). 2017. URL: https://doi.org/10.5281/zenodo.883859.

[17] Nils Werner et al. “trackswitch. js: A Versatile Web-Based Audio Player for Presenting Scientific Results”. In: *Proceedings of the 3rd Web Audio Conference*. 2017.

[4] https://github.com/jpauwels/pywebaudioplayer