Short Tutorial

Import

Import the `simpleaudio` module:

```
import simpleaudio as sa
```

Playing audio directly

The simplest way to play audio is with `play_buffer()`. The `audio_data` parameter must be an object which supports the buffer interface. (`bytes` objects, Python arrays, and **Numpy** arrays all qualify.):

```
play_obj = sa.play_buffer(audio_data, 2, 2, 44100)
```

The *play_obj* object is an instance of `PlayObject` which could be viewed as a ‘handle’ to the audio playback initiated by the `play_buffer()` call. This can be used to stop playback of the audio clip:

```
play_obj.stop()
```

It can used to check whether a sound clip is still playing:

```
if play_obj.is_playing():
    print("still playing")
```

It can also be used to wait for the audio playback to finish. This is espcially useful when a script or program would otherwise exit before playback is done (stopping the playback thread and consequently the audio):

```
play_obj.wait_done()
# script exit
```

WaveObject’s

In order to facilitate cleaner code, the `WaveObject` class is provided which stores a reference to the object containing the audio as well as a copy of the playback parameters. These can be instantiated like so:

```
wave_obj = sa.WaveObject(audio_data, 2, 2, 44100)
```

Playback is started with `play()` and a `PlayObject` is returned as with `play_buffer()`:

```
play_obj = wave_obj.play()
```

A class method exists in order to conveniently create WaveObject instances directly from WAV files on disk:

```
wave_obj = sa.WaveObject.from_wave_file(path_to_file)
```

Similarly, instances can be created from Wave_read objects returned from `wave.open()` from the Python standard library:

```
wave_read = wave.open(path_to_file, 'rb')
wave_obj = sa.WaveObject.from_wave_read(wave_read)
```

Using Numpy

Numpy arrays can be used to store audio but there are a few crucial requirements. If they are to store stereo audio, the array must have two columns since each column contains one channel of audio data. They must also have a signed 16-bit integer dtype and the sample amplitude values must consequently fall in the range of -32768 to 32767. Here is an example of a simple way to ‘normalize’ the audio (making it cover the whole amplitude rage but not exceeding it):

```
audio_array *= 32767 / max(abs(audio_array))
```

And here is an example of converting it to the proper data type (note that this should always be done *after* normalization or other amplitude changes):

```
audio_array = audio_array.astype(np.int16)
```

Here is a full example that plays a few sinewave notes in succession:

```
import  numpy  as np
import simpleaudio as sa

# calculate note frequencies
A_freq = 440
Csh_freq = A_freq * 2 ** (4 / 12)
E_freq = A_freq * 2 ** (7 / 12)

# get timesteps for each sample, T is note duration in seconds
sample_rate = 44100
T = 0.25
t = np.linspace(0, T, T * sample_rate, False)

# generate sine wave notes
A_note = np.sin(A_freq * t * 2 * np.pi)
Csh_note = np.sin(Csh_freq * t * 2 * np.pi)
E_note = np.sin(E_freq * t * 2 * np.pi)

# concatenate notes
audio = np.hstack((A_note, Csh_note, E_note))
# normalize to 16-bit range
audio *= 32767 / np.max(np.abs(audio))
# convert to 16-bit data
audio = audio.astype(np.int16)

# start playback
play_obj = sa.play_buffer(audio, 1, 2, sample_rate)

# wait for playback to finish before exiting
play_obj.wait_done()
```

In order to play stereo audio, the **Numpy** array should have 2 columns. For example, one second of (silent) stereo audio could be produced with:

```
silence = np.zeros((44100, 2))
```

We can then use addition to layer additional audio onto it - in other words, 'mixing' it together. If a signal/audio clip is added to both channels (array columns) equally, then the audio will be perfectly centered and sound just as if it were played in mono. If the proportions vary between the two channels, then the sound will be stronger in one speaker than the other, 'panning' it to one side or the other. The full example below demonstrates this:

```
import  numpy  as np
import simpleaudio as sa

# calculate note frequencies
A_freq = 440
Csh_freq = A_freq * 2 ** (4 / 12)
E_freq = A_freq * 2 ** (7 / 12)

# get timesteps for each sample, T is note duration in seconds
sample_rate = 44100
T = 0.5
t = np.linspace(0, T, T * sample_rate, False)

# generate sine wave notes
A_note = np.sin(A_freq * t * 2 * np.pi)
Csh_note = np.sin(Csh_freq * t * 2 * np.pi)
E_note = np.sin(E_freq * t * 2 * np.pi)

# mix audio together
audio = np.zeros((44100, 2))
n = len(t)
offset = 0
audio[0 + offset: n + offset, 0] += A_note
audio[0 + offset: n + offset, 1] += 0.125 * A_note
offset = 5500
audio[0 + offset: n + offset, 0] += 0.5 * Csh_note
audio[0 + offset: n + offset, 1] += 0.5 * Csh_note
offset = 11000
audio[0 + offset: n + offset, 0] += 0.125 * E_note
audio[0 + offset: n + offset, 1] += E_note

# normalize to 16-bit range
audio *= 32767 / np.max(np.abs(audio))
# convert to 16-bit data
audio = audio.astype(np.int16)

# start playback
play_obj = sa.play_buffer(audio, 2, 2, sample_rate)

# wait for playback to finish before exiting
play_obj.wait_done()
```

24-bit audio can be also be created using **Numpy** but since **Numpy** doesn't have a 24-bit integer dtype, a conversion step is needed. Note also that the max sample value is different for 24-bit audio. A simple (if inefficient) conversion algorithm is demonstrated below, converting an array of 32-bit integers into a `bytes` object which contains the packed 24-bit audio to be played:

```python
import  numpy  as np
import simpleaudio as sa

# calculate note frequencies
A_freq = 440
Csh_freq = A_freq * 2 ** (4 / 12)
E_freq = A_freq * 2 ** (7 / 12)

# get timesteps for each sample, T is note duration in seconds
sample_rate = 44100
T = 0.5
t = np.linspace(0, T, T * sample_rate, False)

# generate sine wave tone
tone = np.sin(440 * t * 2 * np.pi)

# normalize to 24-bit range
tone *= 8388607 / np.max(np.abs(tone))

# convert to 32-bit data
tone = tone.astype(np.int32)

# convert from 32-bit to 24-bit by building a new byte buffer, skipping every fourth
bit
# note: this also works for 2-channel audio
i = 0
byte_array = []
for b in tone.tobytes():
    if i % 4 != 3:
        byte_array.append(b)
    i += 1
audio = bytearray(byte_array)

# start playback
play_obj = sa.play_buffer(audio, 1, 3, sample_rate)

# wait for playback to finish before exiting
play_obj.wait_done()
```