Parselmouth Documentation

Release 0.4.0.dev0

Yannick Jadoul

May 31, 2020

Getting Started

Parselmouth is a Python library for the Praat software.

Though other attempts have been made at porting functionality from Praat to Python, Parselmouth is unique in its aim to provide a complete and Pythonic interface to the internal Praat code. While other projects either wrap Praat's scripting language or reimplementing parts of Praat's functionality in Python, Parselmouth directly accesses Praat's C/C++ code (which means the algorithms and their output are exactly the same as in Praat) and provides efficient access to the program's data, but *also* provides an interface that looks no different from any other Python library.

Please note that Parselmouth is currently in premature state and in active development. While the amount of functionality that is currently present is not huge, more will be added over the next few months. As such, *feedback* and possibly *contributions* are highly appreciated.

Drop by our Gitter chat room or post a message to our Google discussion group if you have any question, remarks, or requests!

Warning: The upcoming release of Parselmouth 0.4.0 will be the *last version* supporting Python 2. Python 2 has reached End Of Life on January 1, 2020, and is officially not supported anymore: see https://python3statement.org/. It is strongly suggested to move to Python 3, to be able to use new Parselmouth functionality after the 0.4.0 release.

CHAPTER 1

Installation

1.1 Basics

Parselmouth can be installed like any other Python library, using (a recent version of) the Python package manager `pip`, on Linux, macOS, and Windows:

```
pip install praat-parselmouth
```

To update your installed version to the latest release, add `-U` (or `--upgrade`) to the command:

```
pip install -U praat-parselmouth
```

Warning: While the Python module itself is called `parselmouth`, the Parselmouth package on the Python Package Index has the name `praat-parselmouth`.

Note: To figure out if you can or should update, the version number of your current Parselmouth installation can be found in the *`parselmouth.VERSION`* variables. The version of Praat on which this version of Parselmouth is based and the release date of that Praat version are available as *`PRAAT_VERSION`* and *`PRAAT_VERSION_DATE`*, respectively.

1.2 Python distributions

Anaconda If you use the Anaconda distribution of Python, you can use the same `pip` command in a terminal of the appropriate Anaconda environment, either activated through the Anaconda Navigator or conda tool.

Homebrew & MacPorts We currently do not have Homebrew or MacPorts packages to install Parselmouth. As far as we know however, Parselmouth can just be installed with the accompanying `pip` of these distributions.

PyPy In priciple, recent versions of PyPy are supported by the pybind11 project and should thus also be supported by Parselmouth. However, we currently have not figured out how to provide precompiled packages, so you will have to still compile the wheel yourself (or contribute an automated way of doing this to the project!).

Other For other distributions of Python, we are expecting that our package is compatible with the Python versions that are out there and that `pip` can handle the installation. If you are using yet another Python distribution, we are definitely interested in hearing about it, so that we can add it to this list!

1.3 PsychoPy

As a Python library, Parselmouth can be used in a PsychoPy experiment. There are two different ways in which PsychoPy can be installed: it can just be manually installed as a standard Python library, in which case Parselmouth can just be installed next to it with `pip`. For Windows and Mac OS X, however, *standalone* versions of PsychoPy exist, and the software does currently not allow for external libraries to be installed with `pip`.

To install Parselmouth in a standalone version of PsychoPy, the following script can be opened and run from within the PsychoPy Coder interface: `psychopy_installation.py`

Note: If running the script results in an error mentioning `TLSV1_ALERT_PROTOCOL_VERSION`, the version of PsychoPy/Python is too old and you will need to follow the manual instructions underneath.

Alternatively, you can follow these steps to manually install Parselmouth into a standalone version of PsychoPy:

0. Find out which version of Python PsychoPy is running.
   - To do so, you can run `import sys; print(sys.version_info)` in the *Shell* tab of the PsychoPy Coder interface. Remember the first two numbers of the version (major and minor; e.g., 3.6).
   - On *Windows*, also run `import platform; print(platform.architecture()[0])` and remember whether the Python executable's architecture is `32bit` or `64bit`.
1. Go to https://pypi.org/project/praat-parselmouth/.
2. Download the file `praat_parselmouth-x.y.z-cpVV-cpVVm-AA.whl` *(for Windows)* or `praat_parselmouth-x.y.z-cpVV-cpVVm-macosx_10_6_intel.whl` *(for Mac OS X)* - where:
   - *x.y.z* will be the version of Parselmouth you want to install
   - *VV* are the first two numbers of the Python version
   - For *Windows*, *AA* is `win32` if you have a `32bit` architecture, and `win_amd64` for `64bit`

   Be sure to find the right file in the list, containing both the correct Python version, and `win32`/`win_amd64` *(Windows)* or `macosx` *(Mac OS X)* in its name!
3. Rename the downloaded file by replacing the `.whl` extension by `.zip`.
4. Extract this zip archive somewhere on your computer, in your directory of choice. Remember the name and location of the extracted folder that contains the file `parselmouth.pyd` *(Windows)* or `parselmouth.so` *(Mac OS X)*.
5. Open PsychoPy, open the *Preferences* window, go to the *General* tab.
6. In the *General* tab of the PsychoPy *Preferences*, in the *paths* field, add the folder where you just extracted the Parselmouth library to the list, enclosing the path in quotemarks. (On *Windows*, also replace all `\` charachters by `/`.)
   - For example, if the list was empty (`[]`), you could make it look like `['C:/Users/Yannick/parselmouth-psychopy/']` or `['/Users/yannick/parselmouth-psychopy/']`.

- On *Windows*, to find the right location to enter in the PsychoPy settings, right click `parselmouth.pyd`, choose *Properties*, and look at the *Location* field.
- On *Mac OS X*, to find the right location to enter in the PsychoPy settings, right click `parselmouth.so`, choose *Get info*, and look at the *where* field.
- On *Mac OS X*, dragging the folder into a terminal window will also give you the full path with slashes.

7. Click *Ok* to save the PsychoPy settings, close the *Preferences* window, and restart PsychoPy.
8. *Optional*: if you want to check if Parselmouth was installed correctly, open the PsychoPy Coder interface, open the *Shell* tab, and type `import parselmouth`.
   - If this results in an error message, please let us know, and we'll try to help you fix what went wrong!
   - If this does not give you an error, congratulations, you can now use Parselmouth in your PsychoPy Builder!

Note: These instructions were tested with the standalone versions 3.1.3 and 1.85.2 of PsychoPy. Things might have changed since then, so if running the script or following the manual steps results in an error, please do not hesitate to get in touch.

1.4 Troubleshooting

It is possible that you run into more problems when trying to install or use Parselmouth. Supporting all of the different Python versions out there is not an easy job, as there are plenty of different platforms and setups.

If you run into problems and these common solutions are not solving them, please drop by the Gitter chat room, write a message in the Google discussion group, create a GitHub issue, or write me a quick email. We would be very happy to solve these problems, so that future users can avoid them!

1.4.1 Multiple Python versions

In case you have multiple installations of Python and don't know which `pip` belongs to which Python version *(looking at you, OS X)*:

```
python -m pip install praat-parselmouth
```

Finding out the exact location of the `python` executable (to call the previous command) for a certain Python installation can be done by typing the following lines in your Python interpreter:

```
>>> import sys
>>> print(sys.executable)
```

If executing this in your Python shell would for example print `/usr/bin/python`, then you would run `/usr/bin/python -m pip install praat-parselmouth` in a terminal to install Parselmouth. (`-U` can again be added to update an already installation to the latest version.)

Combining these two approaches, you can install Parselmouth from within Python itself without knowing where that version of Python is installed:

```
>>> import sys, subprocess
>>> subprocess.call([sys.executable, '-m', 'pip', 'install', 'praat-parselmouth'])
```

Extra arguments to `pip` can be added by inserting them as strings into the list of arguments passed to `subprocess.call` (e.g., to update an existing installation of Parselmouth: `[..., 'install', '-U', 'praat-parselmouth']`).

1.4.2 Pip version

If the standard way to install Parselmouth results in an error or takes a long time, try updating `pip` to the latest version (as `pip` needs to be a reasonably recent version to install the binary, precompiled wheels) by running

```
pip install -U pip
```

If you do not have `pip` installed, you follow these instructions to install pip: https://pip.pypa.io/en/stable/installing/

1.4.3 `ImportError: DLL load failed` on Windows

Sometimes on Windows, the installation works, but importing Parselmouth fails with an error message saying `ImportError: DLL load failed: The specified module could not be found.`. This error is cause by some missing system files, but can luckily be solved quite easily by installing the "Microsoft Visual C++ Redistributable for Visual Studio 2017".

The "Microsoft Visual C++ Redistributable for Visual Studio 2019" installer can be downloaded from Microsoft's website, listed under the "Other Tools and Frameworks" section. These are the direct download links to the relevant files:

- For a 64-bit Python installation: https://aka.ms/vs/16/release/VC_redist.x64.exe
- For a 32-bit Python installation: https://aka.ms/vs/16/release/VC_redist.x86.exe

To check which Python version you are using, you can look at the first line of output when starting a Python shell. The version information should contain `[MSC v.xxxx 64 bit (AMD64)]` in a 64-bit installation, or `[MSC v.xxxx 32 bit (Intel)]` in a 32-bit installation.

CHAPTER 2

Examples

Parselmouth can be used in various contexts to combine Praat functionality with standard Python features or other Python libraries. The following examples give an idea of the range of possibilities:

2.1 Plotting

Using Parselmouth, it is possible to use the existing Python plotting libraries – such as Matplotlib and seaborn – to make custom visualizations of the speech data and analysis results obtained by running Praat's algorithms.

The following examples visualize an audio recording of someone saying *"The north wind and the sun [...]"*: the_north_wind_and_the_sun.wav, extracted from a Wikipedia Commons audio file.

We start out by importing `parselmouth`, some common Python plotting libraries `matplotlib` and `seaborn`, and the `numpy` numeric library.

[1]:
```
import parselmouth

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

[2]:
```
sns.set() # Use seaborn's default style to make attractive graphs
plt.rcParams['figure.dpi'] = 100 # Show nicely large images in this notebook
```

Once we have the necessary libraries for this example, we open and read in the audio file and plot the raw waveform.

[3]:
```
snd = parselmouth.Sound("audio/the_north_wind_and_the_sun.wav")
```

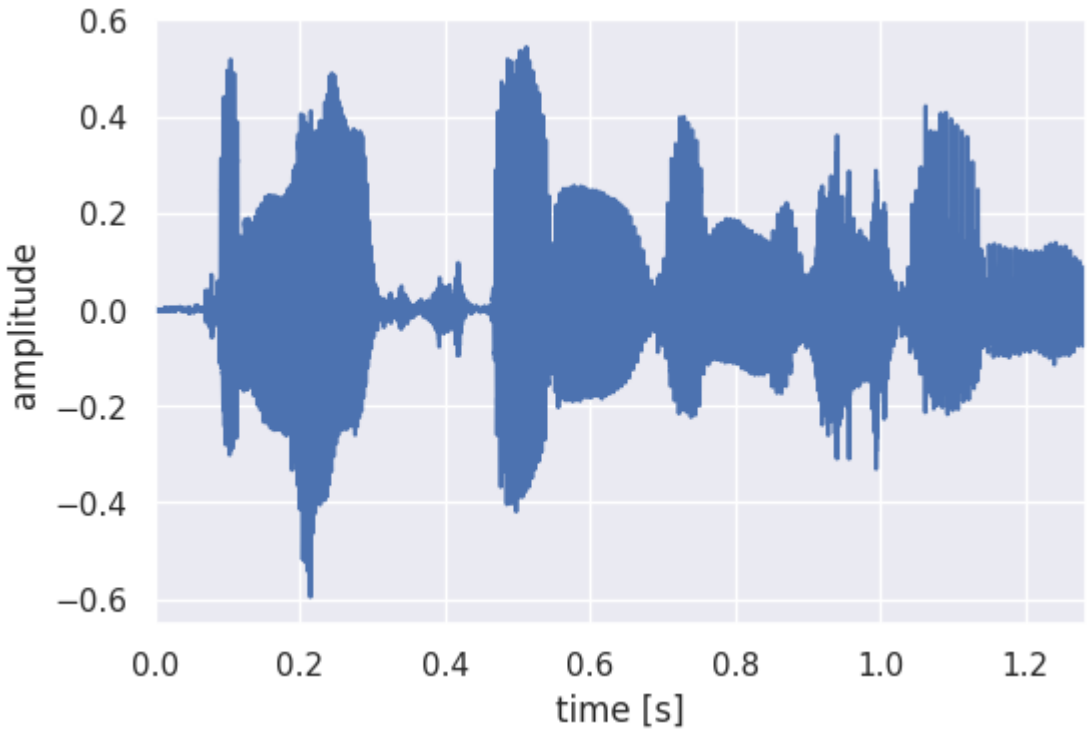
`snd` is now a Parselmouth *Sound* object, and we can access its values and other properties to plot them with the common `matplotlib` Python library:

[4]:
```
plt.figure()
plt.plot(snd.xs(), snd.values.T)
```

(continues on next page)

(continued from previous page)

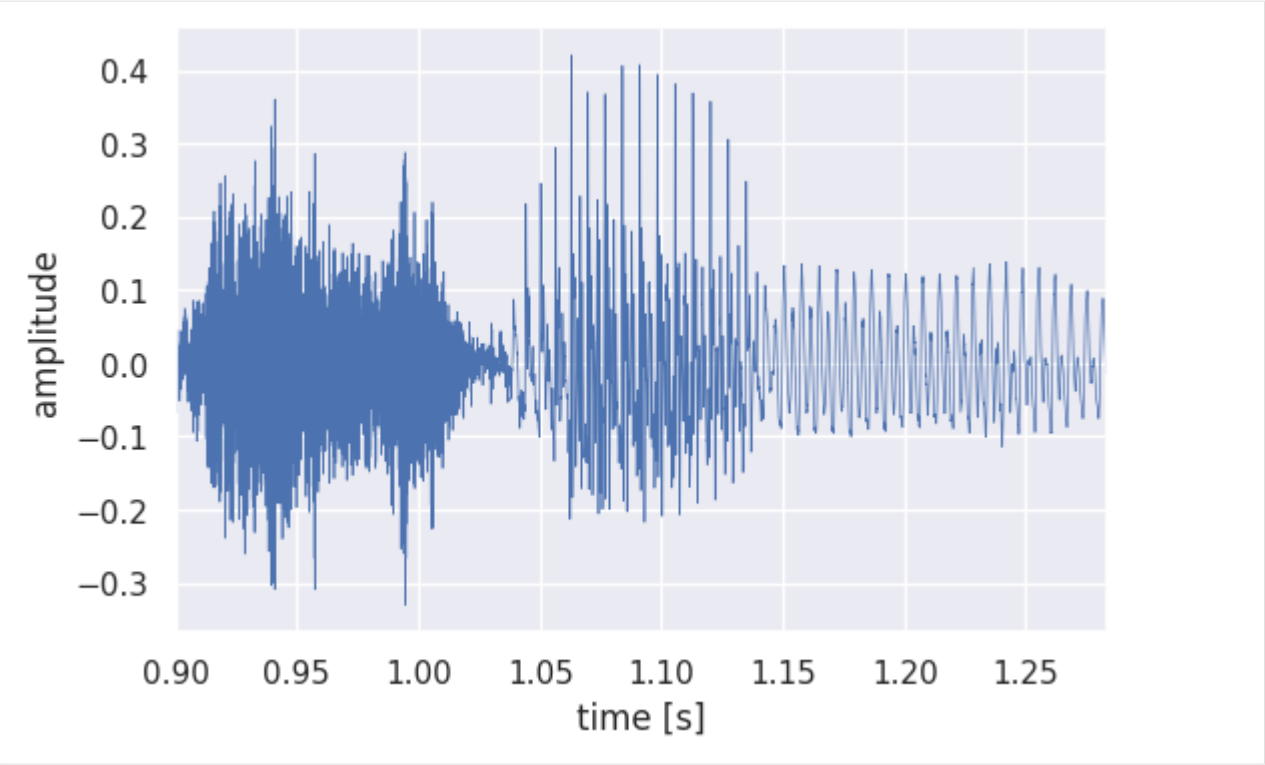
```
plt.xlim([snd.xmin, snd.xmax])
plt.xlabel("time [s]")
plt.ylabel("amplitude")
plt.show() # or plt.savefig("sound.png"), or plt.savefig("sound.pdf")
```



It is also possible to extract part of the speech fragment and plot it separately. For example, let's extract the word *"sun"* and plot its waveform with a finer line.

[5]:
```
snd_part = snd.extract_part(from_time=0.9, preserve_times=True)
```

[6]:
```
plt.figure()
plt.plot(snd_part.xs(), snd_part.values.T, linewidth=0.5)
plt.xlim([snd_part.xmin, snd_part.xmax])
plt.xlabel("time [s]")
plt.ylabel("amplitude")
plt.show()
```



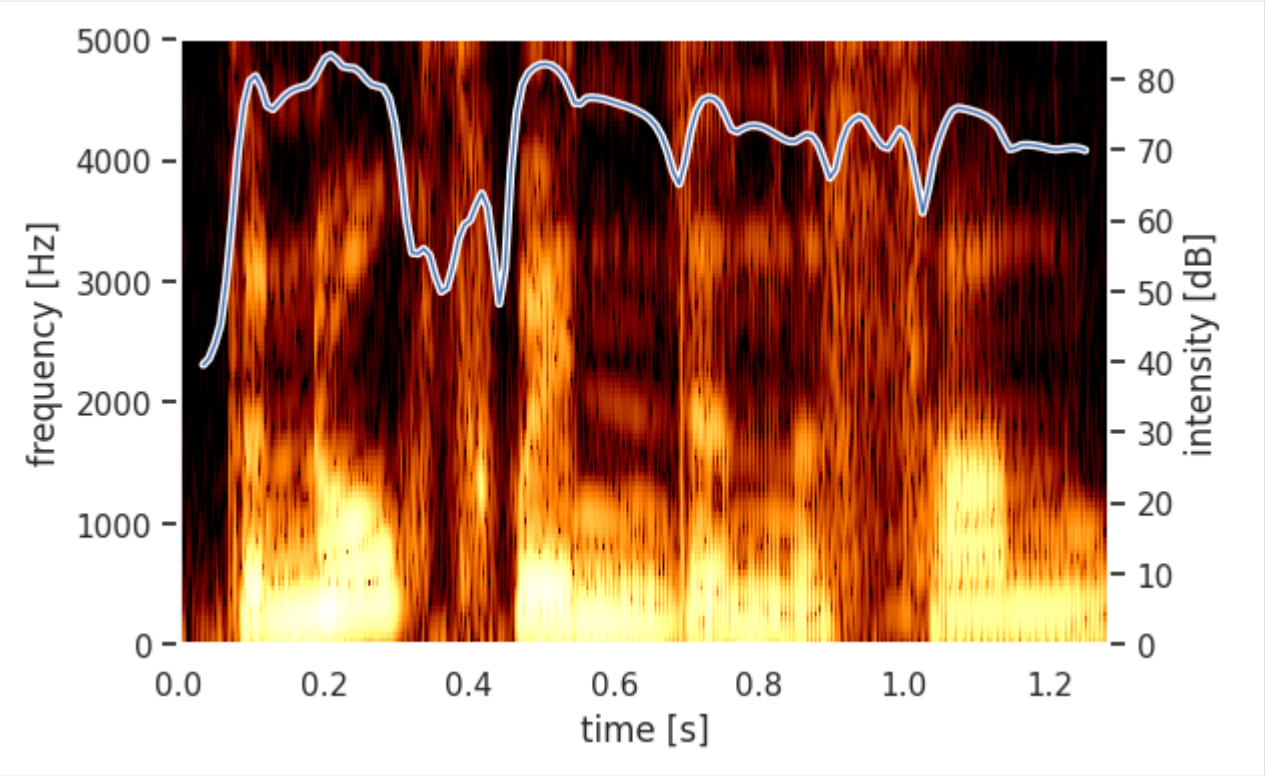
Next, we can write a couple of ordinary Python functions to plot a Parselmouth `Spectrogram` and `Intensity`.

[7]:
```
def draw_spectrogram(spectrogram, dynamic_range=70):
    X, Y = spectrogram.x_grid(), spectrogram.y_grid()
    sg_db = 10 * np.log10(spectrogram.values)
    plt.pcolormesh(X, Y, sg_db, vmin=sg_db.max() - dynamic_range, cmap='afmhot')
    plt.ylim([spectrogram.ymin, spectrogram.ymax])
    plt.xlabel("time [s]")
    plt.ylabel("frequency [Hz]")

def draw_intensity(intensity):
    plt.plot(intensity.xs(), intensity.values.T, linewidth=3, color='w')
    plt.plot(intensity.xs(), intensity.values.T, linewidth=1)
    plt.grid(False)
    plt.ylim(0)
    plt.ylabel("intensity [dB]")
```

After defining how to plot these, we use Praat (through Parselmouth) to calculate the spectrogram and intensity to actually plot the intensity curve overlaid on the spectrogram.

[8]:
```
intensity = snd.to_intensity()
spectrogram = snd.to_spectrogram()
plt.figure()
draw_spectrogram(spectrogram)
plt.twinx()
draw_intensity(intensity)
plt.xlim([snd.xmin, snd.xmax])
plt.show()
```



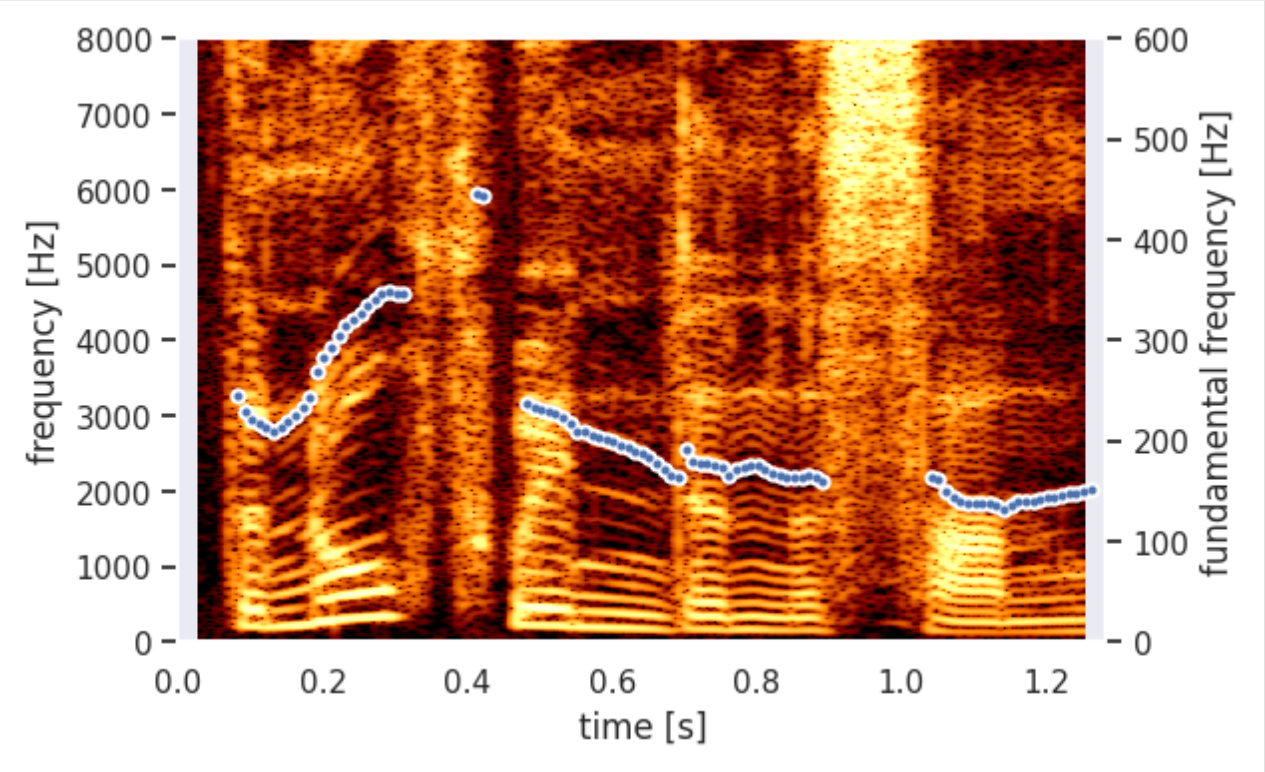
The Parselmouth functions and methods have the same arguments as the Praat commands, so we can for example also change the window size of the spectrogram analysis to get a narrow-band spectrogram. Next to that, let's now have Praat calculate the pitch of the fragment, so we can plot it instead of the intensity.

[9]:
```
def draw_pitch(pitch):
    # Extract selected pitch contour, and
    # replace unvoiced samples by NaN to not plot
    pitch_values = pitch.selected_array['frequency']
    pitch_values[pitch_values==0] = np.nan
    plt.plot(pitch.xs(), pitch_values, 'o', markersize=5, color='w')
    plt.plot(pitch.xs(), pitch_values, 'o', markersize=2)
    plt.grid(False)
    plt.ylim(0, pitch.ceiling)
    plt.ylabel("fundamental frequency [Hz]")
```

[10]:
```
pitch = snd.to_pitch()
```

[11]:
```
# If desired, pre-emphasize the sound fragment before calculating the spectrogram
pre_emphasized_snd = snd.copy()
pre_emphasized_snd.pre_emphasize()
spectrogram = pre_emphasized_snd.to_spectrogram(window_length=0.03, maximum_
↪frequency=8000)
```

[12]:
```
plt.figure()
draw_spectrogram(spectrogram)
plt.twinx()
draw_pitch(pitch)
plt.xlim([snd.xmin, snd.xmax])
plt.show()
```



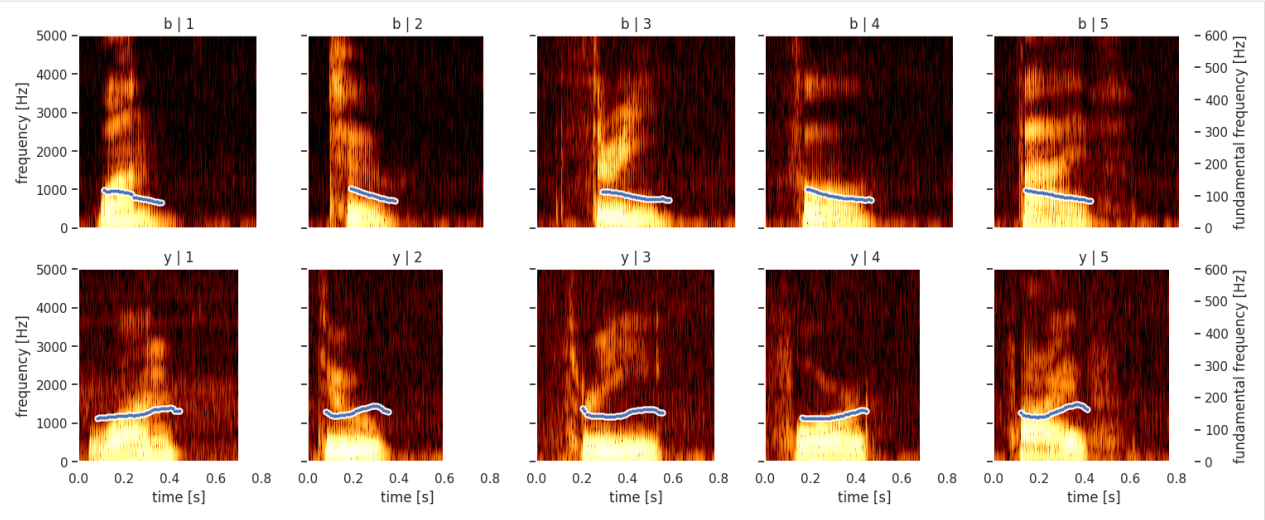
Using the FacetGrid functionality from `seaborn`, we can even plot plot multiple a structured grid of multiple custom spectrograms. For example, we will read a CSV file (using the pandas library) that contains the digit that was spoken, the ID of the speaker and the file name of the audio fragment: digit_list.csv, 1_b.wav, 2_b.wav, 3_b.wav, 4_b.wav, 5_b.wav, 1_y.wav, 2_y.wav, 3_y.wav, 4_y.wav, 5_y.wav

[13]:
```
import pandas as pd

def facet_util(data, **kwargs):
    digit, speaker_id = data[['digit', 'speaker_id']].iloc[0]
    sound = parselmouth.Sound("audio/{}_{}.wav".format(digit, speaker_id))
    draw_spectrogram(sound.to_spectrogram())
    plt.twinx()
    draw_pitch(sound.to_pitch())
    # If not the rightmost column, then clear the right side axis
    if digit != 5:
        plt.ylabel("")
        plt.yticks([])

results = pd.read_csv("other/digit_list.csv")

grid = sns.FacetGrid(results, row='speaker_id', col='digit')
grid.map_dataframe(facet_util)
grid.set_titles(col_template="{col_name}", row_template="{row_name}")
grid.set_axis_labels("time [s]", "frequency [Hz]")
grid.set(facecolor='white', xlim=(0, None))
plt.show()
```



2.2 Batch processing of files

Using the Python standard libraries (i.e., the `glob` and `os` modules), we can also quickly code up batch operations e.g. over all files with a certain extension in a directory. For example, we can make a list of all `.wav` files in the `audio` directory, use Praat to pre-emphasize these *Sound* objects, and then write the pre-emphasized sound to a `WAV` and `AIFF` format file.

[1]:
```
# Find all .wav files in a directory, pre-emphasize and save as new .wav and .aiff␣
↪file
import parselmouth

import glob
import os.path

for wave_file in glob.glob("audio/*.wav"):
    print("Processing {}...".format(wave_file))
    s = parselmouth.Sound(wave_file)
    s.pre_emphasize()
    s.save(os.path.splitext(wave_file)[0] + "_pre.wav", 'WAV') # or parselmouth.
↪SoundFileFormat.WAV instead of 'WAV'
    s.save(os.path.splitext(wave_file)[0] + "_pre.aiff", 'AIFF')
```

```
Processing audio/3_y.wav...
Processing audio/5_y.wav...
Processing audio/the_north_wind_and_the_sun.wav...
Processing audio/4_y.wav...
Processing audio/2_y.wav...
Processing audio/5_b.wav...
Processing audio/bat.wav...
Processing audio/3_b.wav...
Processing audio/1_y.wav...
Processing audio/bet.wav...
Processing audio/2_b.wav...
Processing audio/1_b.wav...
Processing audio/4_b.wav...
```

After running this, the original home directory now contains all of the original `.wav` files pre-emphazised and written

again as `.wav` and `.aiff` files. The reading, pre-emphasis, and writing are all done by Praat, while looping over all `.wav` files is done by standard Python code.

[2]:
```
# List the current contents of the audio/ folder
!ls audio/
```

```
1_b.wav       2_y_pre.aiff  4_b_pre.wav   bat.wav
1_b_pre.aiff  2_y_pre.wav   4_y.wav          bat_pre.aiff
1_b_pre.wav   3_b.wav            4_y_pre.aiff  bat_pre.wav
1_y.wav       3_b_pre.aiff  4_y_pre.wav   bet.wav
1_y_pre.aiff  3_b_pre.wav   5_b.wav          bet_pre.aiff
1_y_pre.wav   3_y.wav            5_b_pre.aiff  bet_pre.wav
2_b.wav       3_y_pre.aiff  5_b_pre.wav   the_north_wind_and_the_sun.wav
2_b_pre.aiff  3_y_pre.wav   5_y.wav          the_north_wind_and_the_sun_pre.aiff
2_b_pre.wav   4_b.wav            5_y_pre.aiff  the_north_wind_and_the_sun_pre.wav
2_y.wav       4_b_pre.aiff  5_y_pre.wav
```

[3]:
```
# Remove the generated audio files again, to clean up the output from this example
!rm audio/*_pre.wav
!rm audio/*_pre.aiff
```

Similarly, we can use the pandas library to read a CSV file with data collected in an experiment, and loop over that data to e.g. extract the mean harmonics-to-noise ratio. The `results` CSV has the following structure:

condition	...	pp_id
0	...	1877
1	...	801
1	...	2456
0	...	3126

The following code would read such a table, loop over it, use Praat through Parselmouth to calculate the analysis of each row, and then write an augmented CSV file to disk. To illustrate we use an example set of sound fragments: results.csv, 1_b.wav, 2_b.wav, 3_b.wav, 4_b.wav, 5_b.wav, 1_y.wav, 2_y.wav, 3_y.wav, 4_y.wav, 5_y.wav

In our example, the original CSV file, results.csv contains the following table:

[4]:
```
import pandas as pd

print(pd.read_csv("other/results.csv"))
```

```
   condition pp_id
0          3     y
1          5     y
2          4     b
3          2     y
4          5     b
5          2     b
6          3     b
7          1     y
8          1     b
9          4     y
```

[5]:
```
def analyse_sound(row):
    condition, pp_id = row['condition'], row['pp_id']
    filepath = "audio/{}_{}.wav".format(condition, pp_id)
    sound = parselmouth.Sound(filepath)
    harmonicity = sound.to_harmonicity()
```

(continues on next page)

(continued from previous page)

```
    return harmonicity.values[harmonicity.values != -200].mean()

# Read in the experimental results file
dataframe = pd.read_csv("other/results.csv")

# Apply parselmouth wrapper function row-wise
dataframe['harmonics_to_noise'] = dataframe.apply(analyse_sound, axis='columns')

# Write out the updated dataframe
dataframe.to_csv("processed_results.csv", index=False)
```

We can now have a look at the results by reading in the `processed_results.csv` file again:

[6]:
```
print(pd.read_csv("processed_results.csv"))
```

```
   condition pp_id  harmonics_to_noise
0          3     y           22.615414
1          5     y           16.403205
2          4     b           17.839167
3          2     y           21.054674
4          5     b           16.092489
5          2     b           12.378289
6          3     b           15.718858
7          1     y           16.704779
8          1     b           12.874451
9          4     y           18.431586
```

[7]:
```
# Clean up, remove the CSV file generated by this example
!rm processed_results.csv
```

2.3 Pitch manipulation and Praat commands

Another common use of Praat functionality is to manipulate certain features of an existing audio fragment. For example, in the context of a perception experiment one might want to change the pitch contour of an existing audio stimulus while keeping the rest of the acoustic features the same. Parselmouth can then be used to access the Praat algorithms that accompish this, from Python.

Since this Praat `Manipulation` functionality has currently not been ported to Parselmouth's Python interface, we will need to use Parselmouth interface to access *raw* Praat commands.

In this example, we will increase the pitch contour of an audio recording of the word *"four"*, 4_b.wav, by one octave. To do so, let's start by importing Parselmouth and opening the audio file:

[1]:
```
import parselmouth

sound = parselmouth.Sound("audio/4_b.wav")
```

We can also listen to this audio fragment:

[2]:
```
from IPython.display import Audio
Audio(data=sound.values, rate=sound.sampling_frequency)
```

[2]:
```
<IPython.lib.display.Audio object>
```

However, now we want to use the Praat `Manipulation` functionality, but unfortunately, Parselmouth does not yet contain a `Manipulation` class and the necessary functionality is not directly accessible through the *Sound*

object `sound`. To directly access the Praat commands conveniently from Python, we can make use of the *parselmouth.praat.call* function.

[3]:
```
from parselmouth.praat import call

manipulation = call(sound, "To Manipulation", 0.01, 75, 600)
```

[4]:
```
type(manipulation)
```

[4]:
```
parselmouth.Data
```

Note how we first pass in the object(s) that would be selected in Praat's object list. The next argument to this function is the name of the command as it would be used in a script or can be seen in the Praat user interface. Finally, the arguments to this command's parameters are passed to the function (in this case, Praat's default values for *"Time step (s)"*, *"Minimum pitch (Hz)"*, and *"Maximum pitch (Hz)"*). This call to `parselmouth.praat.call` will then return the result of the command as a Python type or Parselmouth object. In this case, a Praat `Manipulation` object would be created, so our function returns a `parselmouth.Data` object, as a `parselmouth.Manipulation` class does not exist in Parselmouth. However, we can still query the class name the underlying Praat object has:

[5]:
```
manipulation.class_name
```

[5]:
```
'Manipulation'
```

Next, we can continue using Praat functionality to further use this `manipulation` object similar to how one would achieve this in Praat. Here, note how we can mix normal Python (e.g. integers and lists), together with the normal use of Parselmouth as Python library (e.g., `sound.xmin`) as well as with the `parselmouth.praat.call` function.

[6]:
```
pitch_tier = call(manipulation, "Extract pitch tier")

call(pitch_tier, "Multiply frequencies", sound.xmin, sound.xmax, 2)

call([pitch_tier, manipulation], "Replace pitch tier")
sound_octave_up = call(manipulation, "Get resynthesis (overlap-add)")
```

[7]:
```
type(sound_octave_up)
```

[7]:
```
parselmouth.Sound
```

The last invocation of `call` resulted in a Praat `Sound` object being created and returned. Because Parselmouth knows that this type corresponds to a `parselmouth.Sound` Python object, the Python type of this object is not a `parselmouth.Data`. Rather, this object is now equivalent to the one we created at the start of this example. As such, we can use this new object normally, calling methods and accessing its contents. Let's listen and see if we succeeded in increasing the pitch by one octave:

[8]:
```
Audio(data=sound_octave_up.values, rate=sound_octave_up.sampling_frequency)
```

[8]:
```
<IPython.lib.display.Audio object>
```

And similarly, we could also for example save the sound to a new file.

[9]:
```
sound_octave_up.save("4_b_octave_up.wav", "WAV")
```

[10]:
```
Audio(filename="4_b_octave_up.wav")
```

[10]:
```
<IPython.lib.display.Audio object>
```

[11]:
```
# Clean up the created audio file again
!rm 4_b_octave_up.wav
```

We can of course also turn this combination of commands into a custom function, to be reused in later code:

[12]:
```
def change_pitch(sound, factor):
    manipulation = call(sound, "To Manipulation", 0.01, 75, 600)

    pitch_tier = call(manipulation, "Extract pitch tier")

    call(pitch_tier, "Multiply frequencies", sound.xmin, sound.xmax, factor)

    call([pitch_tier, manipulation], "Replace pitch tier")
    return call(manipulation, "Get resynthesis (overlap-add)")
```

Using Jupyter widgets, one can then change the audio file or the pitch change factor, and interactively hear how this sounds.

To try this for yourself, open an online, interactive version of this notebook on Binder! (see link at the top of this notebook)

[13]:
```
import ipywidgets
import glob

def interactive_change_pitch(audio_file, factor):
    sound = parselmouth.Sound(audio_file)
    sound_changed_pitch = change_pitch(sound, factor)
    return Audio(data=sound_changed_pitch.values, rate=sound_changed_pitch.sampling_
↪frequency)

w = ipywidgets.interact(interactive_change_pitch,
                        audio_file=ipywidgets.Dropdown(options=sorted(glob.glob(
↪"audio/*.wav")), value="audio/4_b.wav"),
                        factor=ipywidgets.FloatSlider(min=0.25, max=4, step=0.05,␣
↪value=1.5))
```

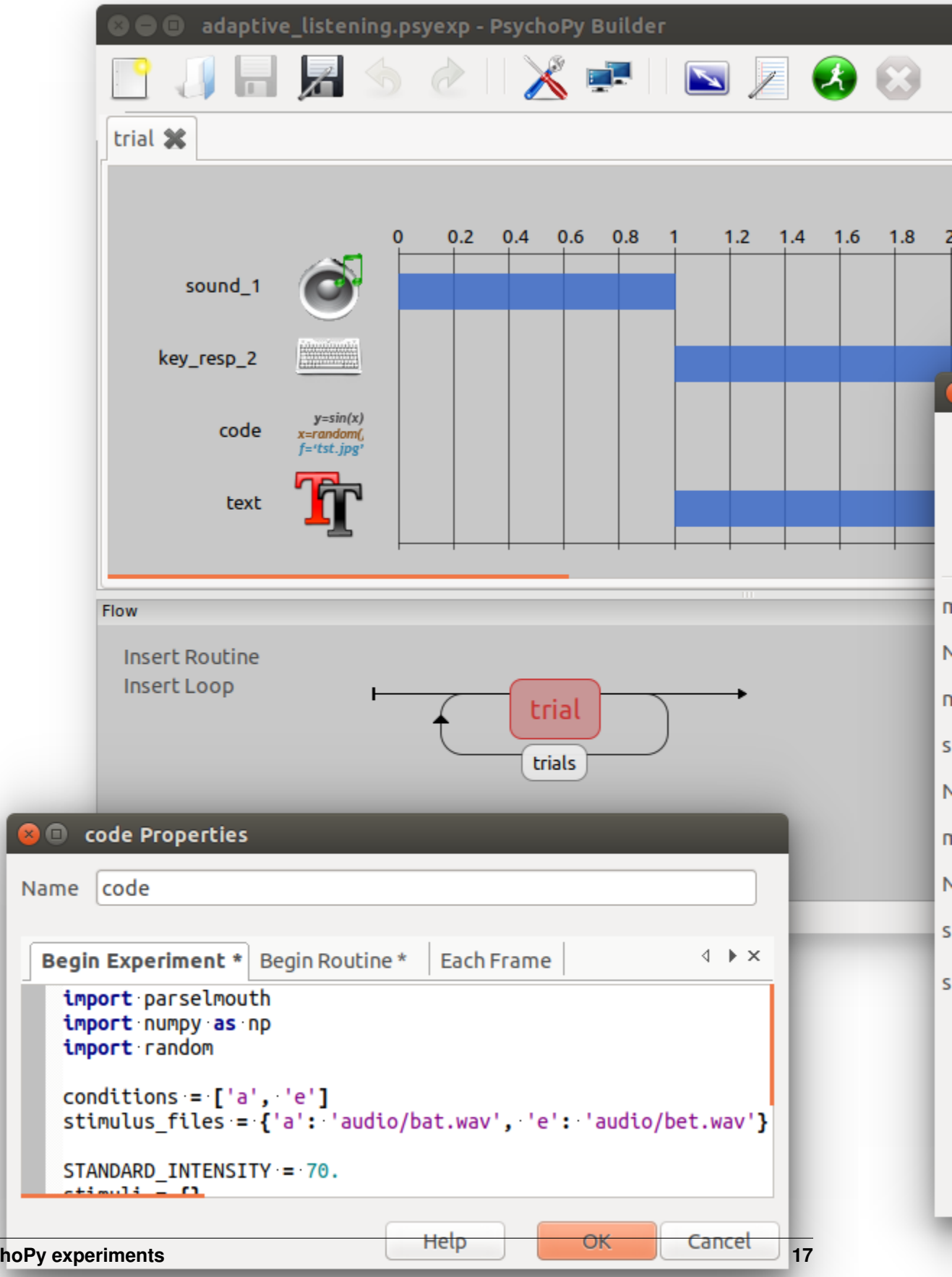
```
<IPython.lib.display.Audio object>
```

2.4 PsychoPy experiments

Parselmouth also allows Praat functionality to be included in an interactive PsychoPy experiment (refer to the subsection on *installing Parselmouth for PsychoPy* for detailed installation instructions for the PsychoPy graphical interface, the *PsychoPy Builder*). The following example shows how easily Python code that uses Parselmouth can be injected in such an experiment; following an adaptive staircase experimental design, at each trial of the experiment a new stimulus is generated based on the responses of the participant. See e.g. Kaernbach, C. (2001). Adaptive threshold estimation with unforced-choice tasks. *Attention, Perception, & Psychophysics*, *63*, 1377–1388., or the PsychoPy tutorial at https://www.psychopy.org/coder/tutorial2.html.

In this example, we use an adaptive staircase experiment to determine the minimal amount of noise that makes the participant unable to distinguish between two audio fragments, *"bat"* and *"bet"* (bat.wav, bet.wav). At every iteration of the experiment, we want to generate a version of these audio files with a specific signal-to-noise ratio, of course using Parselmouth to do so. Depending on whether the participant correctly identifies whether the noisy stimulus was *"bat"* or *"bet"*, the noise level is then either increased or decreased.

As Parselmouth is just another Python library, using it from the PsychoPy *Coder* interface or from a standard Python script that imports the `psychopy` module is quite straightforward. However, PsychoPy also features a so-called *Builder* interface, which is a graphical interface to set up experiments with minimal or no coding. In this *Builder*, a user can create multiple experimental *'routines'* out of different *'components'* and combine them through *'loops'*, that can all be configured graphically:


adaptive_listening.psyexp - PsychoPy Builder
trial
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
sound_1
key_resp_2
code
y=sin(x)
x=random(,
f='tst.jpg'
text
Flow
Insert Routine
Insert Loop
trial
trials
code Properties
Name
code
Begin Experiment *
Begin Routine *
Each Frame
import parselmouth
import numpy as np
import random
conditions = ['a', 'e']
stimulus_files = {'a': 'audio/bat.wav', 'e': 'audio/bet.wav'}
STANDARD_INTENSITY = 70.
Help
OK
Cancel

For our simple example, we create a single routine `trial`, with a `Sound`, a `Keyboard`, and a `Text` component. We also insert a loop around this routine of the type `staircase`, such that PsychoPy will take care of the actual implementation of the loop in adaptive staircase design. The full PsychoPy experiment which can be opened in the *Builder* can be downloaded here: adaptive_listening.psyexp

Finally, to customize the behavior of the `trial` routine and to be able to use Parselmouth inside the PsychoPy experiment, we still add a `Code` component to the routine. This component will allow us to write Python code that interacts with the rest of the components and with the adaptive staircase loop. The `Code` components has different tabs, that allow us to insert custom code at different points during the execution of our trial.

First, there is the **Begin Experiment** tab. The code in this tab is executed only once, at the start of the experiment. We use this to set up the Python environment, importing modules and initializing variables, and defining constants:

[1]:
```
# ** Begin Experiment **

import parselmouth
import numpy as np
import random

conditions = ['a', 'e']
stimulus_files = {'a': "audio/bat.wav", 'e': "audio/bet.wav"}

STANDARD_INTENSITY = 70.
stimuli = {}
for condition in conditions:
    stimulus = parselmouth.Sound(stimulus_files[condition])
    stimulus.scale_intensity(STANDARD_INTENSITY)
    stimuli[condition] = stimulus
```

The code in the **Begin Routine** tab is executed before the routine, so in our example, for every iteration of the surrounding staircase loop. This allows us to actually use Parselmouth to generate the stimulus that should be played to the participant during this iteration of the routine. To do this, we need to access the current value of the adaptive staircase algorithm: PsychoPy stores this in the Python variable `level`. For example, at some point during the experiment, this could be 10 (representing a signal-to-noise ratio of 10 dB):

[2]:
```
level = 10
```

To execute the code we want to put in the **Begin Routine** tab, we need to add a few variables that would be made available by the PsychoPy Builder, normally:

[3]:
```
# 'filename' variable is also set by PsychoPy and contains base file name of saved␣
↪log/output files
filename = "data/participant_staircase_23032017"

# PsychoPy also create a Trials object, containing e.g. information about the current␣
↪iteration of the loop
# So let's quickly fake this, in this example, such that the code can be executed␣
↪without errors
# In PsychoPy this would be a `psychopy.data.TrialHandler` (https://www.psychopy.org/
↪api/data.html#psychopy.data.TrialHandler)
class MockTrials:
    def addResponse(self, response):
        print("Registering that this trial was {}successful".format("" if response␣
↪else "un"))
trials = MockTrials()
trials.thisTrialN = 5 # We only need the 'thisTrialN' attribute of the 'trials'␣
↪variable
```

(continues on next page)

(continued from previous page)

```
# The Sound component can also be accessed by it's name, so let's quickly mock that␣
↪as well
# In PsychoPy this would be a `psychopy.sound.Sound` (https://www.psychopy.org/api/
↪sound.html#psychopy.sound.Sound)
class MockSound:
    def setSound(self, file_name):
        print("Setting audio file of Sound component to '{}'".format(file_name))
sound_1 = MockSound()

# And the same for our Keyboard component, `key_resp_2`:
class MockKeyboard:
    pass
key_resp_2 = MockKeyboard()

# Finally, let's also seed the random module to have a consistent output across␣
↪different runs
random.seed(42)
```

[4]:
```
# Let's also create the directory where we will store our example output
!mkdir data
```

Now, we can execute the code that would be in the **Begin Routine** tab:

[5]:
```
# ** Begin Routine **

random_condition = random.choice(conditions)
random_stimulus = stimuli[random_condition]

noise_samples = np.random.normal(size=random_stimulus.n_samples)
noisy_stimulus = parselmouth.Sound(noise_samples,
                    sampling_frequency=random_stimulus.sampling_frequency)
noisy_stimulus.scale_intensity(STANDARD_INTENSITY - level)
noisy_stimulus.values += random_stimulus.values
noisy_stimulus.scale_intensity(STANDARD_INTENSITY)

# use 'filename' to save our custom stimuli
stimulus_file_name = filename + "_stimulus_" + str(trials.thisTrialN) + ".wav"
noisy_stimulus.resample(44100).save(stimulus_file_name, 'WAV')
sound_1.setSound(stimulus_file_name)
```

```
Setting audio file of Sound component to 'data/participant_staircase_23032017_
↪stimulus_5.wav'
```

Let's listen to the file we have just generated and that we would play to the participant:

[6]:
```
from IPython.display import Audio
Audio(filename="data/participant_staircase_23032017_stimulus_5.wav")
```

[6]:
```
<IPython.lib.display.Audio object>
```

In this example, we do not really need to have code executed during the trial (i.e., in the **Each Frame** tab). However, at the end of the trial, we need to inform the PsychoPy staircase loop whether the participant was correct or not, because this will affect the further execution the adaptive staircase, and thus value of the `level` variable set by PsychoPy. For this we add a final line in the **End Routine** tab. Let's say the participant guessed *"bat"* and pressed the `a` key:

[7]:
```
key_resp_2.keys = 'a'
```

The **End Routine** tab then contains the following code to check the participant's answer against the randomly chosen condition, and to inform the `trials` object of whether the participant was correct:

[8]:
```
# ** End Routine **

trials.addResponse(key_resp_2.keys == random_condition)
```

```
Registering that this trial was successful
```

[9]:
```
# Clean up the output directory again
!rm -r data
```

2.5 Web service

Since Parselmouth is a normal Python library, it can also easily be used within the context of a web server. There are several Python frameworks that allow to quickly set up a web server or web service. In this examples, we will use Flask to show how easily one can set up a web service that uses Parselmouth to access Praat functionality such as the pitch track estimation algorithms. This functionality can then be accessed by clients without requiring either Praat, Parselmouth, or even Python to be installed, for example within the context of an online experiment.

All that is needed to set up the most basic web server in Flask is a single file. We adapt the standard Flask example to accept a sound file, access Parselmouth's *Sound.to_pitch*, and then send back the list of pitch track frequencies. Note that apart from saving the file that was sent in the HTTP request and encoding the resulting list of frequencies in JSON, the Python code of the `pitch_track` function is the same as one would write in a normal Python script using Parselmouth.

[1]:
```
%%writefile server.py

from flask import Flask, request, jsonify
import tempfile

app = Flask(__name__)

@app.route('/pitch_track', methods=['POST'])
def pitch_track():
    import parselmouth

    # Save the file that was sent, and read it into a parselmouth.Sound
    with tempfile.NamedTemporaryFile() as tmp:
        tmp.write(request.files['audio'].read())
        sound = parselmouth.Sound(tmp.name)

    # Calculate the pitch track with Parselmouth
    pitch_track = sound.to_pitch().selected_array['frequency']

    # Convert the NumPy array into a list, then encode as JSON to send back
    return jsonify(list(pitch_track))
```

```
Writing server.py
```

Normally, we can then run the server typing `FLASK_APP=server.py flask run` on the command line, as explained in the Flask documentation. Please do note that to run this server publicly, in a secure way and as part of a bigger setup, other options are available to deploy! Refer to the Flask deployment documentation.

However, to run the server from this Jupyter notebook and still be able to run the other cells that access the functionality on the client side, the following code will start the server in a separate thread and print the output of the running server.

[2]:
```
import os
import subprocess
import sys
import time

# Start a subprocess that runs the Flask server
p = subprocess.Popen([sys.executable, "-m", "flask", "run"], env=dict(**os.environ,␣
↪FLASK_APP="server.py"), stdout=subprocess.PIPE, stderr=subprocess.PIPE)

# Start two subthreads that forward the output from the Flask server to the output of␣
↪the Jupyter notebook
def forward(i, o):
    while p.poll() is None:
        l = i.readline().decode('utf-8')
        if l:
            o.write("[SERVER] " + l)

import threading
threading.Thread(target=forward, args=(p.stdout, sys.stdout)).start()
threading.Thread(target=forward, args=(p.stderr, sys.stderr)).start()

# Let's give the server a bit of time to make sure it has started
time.sleep(2)
```

```
[SERVER]  * Serving Flask app "server.py"
[SERVER]  * Environment: production
[SERVER]    WARNING: This is a development server. Do not use it in a production␣
↪deployment.
[SERVER]    Use a production WSGI server instead.
[SERVER]  * Debug mode: off
```

```
[SERVER]  * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Now that the server is up and running, we can make a standard HTTP request to this web service. For example, we can send a Wave file with an audio recording of someone saying *"The north wind and the sun [...]"*: the_north_wind_and_the_sun.wav, extracted from a Wikipedia Commons audio file.

[3]:
```
from IPython.display import Audio
Audio(filename="audio/the_north_wind_and_the_sun.wav")
```

[3]:
```
<IPython.lib.display.Audio object>
```

To do so, we use the requests library in this example, but we could use any library to send a standard HTTP request.

[4]:
```
import requests
import json

# Load the file to send
files = {'audio': open("audio/the_north_wind_and_the_sun.wav", 'rb')}
# Send the HTTP request and get the reply
reply = requests.post("http://127.0.0.1:5000/pitch_track", files=files)
# Extract the text from the reply and decode the JSON into a list
pitch_track = json.loads(reply.text)
print(pitch_track)
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 245.46350823838623, 228.46732331898713, 220.
↪22988190492924, 217.9494117767135, 212.3212009488913, 208.4237107759689, 213.
↪32102922639103, 219.221641698201, 225.08564349512903, 232.58018422224498, 243.
↪61028546789302, 267.9586673912011, 283.57192373333476, 293.0908779503718, 303.
↪9716558499233, 314.16812500380297, 320.11744147538917, 326.343950137746, 333.
↪36323872999395, 340.0277922284553, 345.82407490135375, 348.57743419008335, 346.
↪9665344071829, 346.5317932196539, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 445.
↪13555899369443, 442.9936784742193, 0.0, 0.0, 0.0, 0.0, 0.0, 236.39129492565098, 233.
↪77304383723657, 231.61759184149392, 229.25293731480636, 226.5388725505906, 223.
↪6713912523249, 217.56247158178115, 208.75233223927862, 208.36854272328 42, 205.
↪1132684638252, 202.9962832843818, 200.7424552975565, 198.37924372325074, 195.
↪7138772191214, 192.92640662274337, 189.55087006227458, 186.29856998961736, 182.
```

(continues on next page)

(continued from previous page)

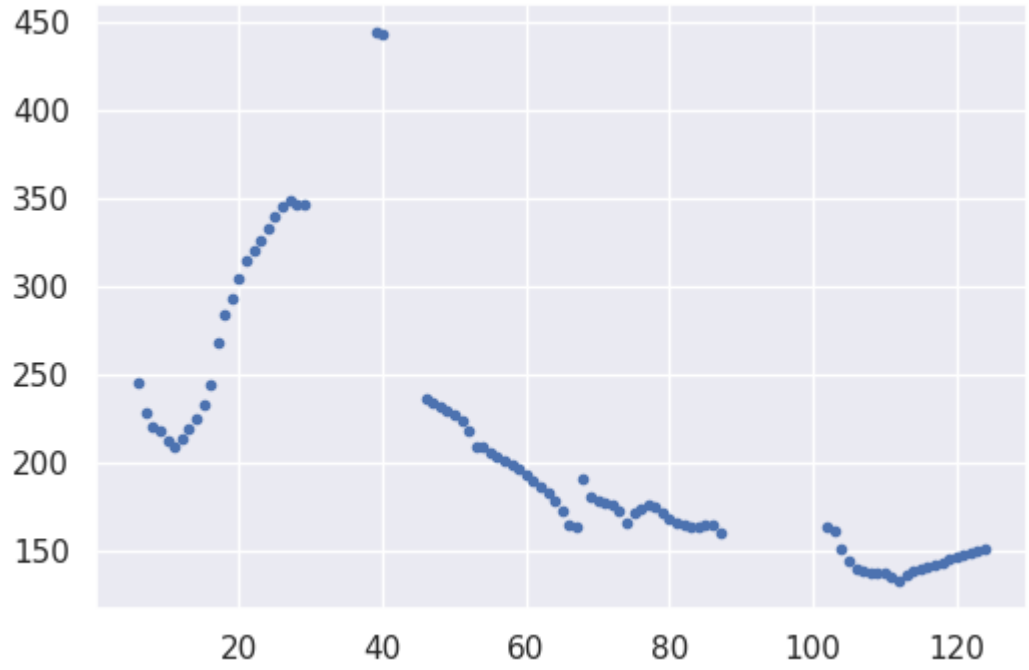
```
[SERVER] 127.0.0.1 - - [31/May/2020 21:54:35] "POST /pitch_track HTTP/1.1" 200 -
```

Since we used the standard `json` library from Python to decode the reply from server, `pitch_track` is now a normal list of `floats` and we can for example plot the estimated pitch track:

[5]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

[6]:
```
sns.set() # Use seaborn's default style to make attractive graphs
plt.rcParams['figure.dpi'] = 100 # Show nicely large images in this notebook
```

[7]:
```
plt.figure()
plt.plot([float('nan') if x == 0.0 else x for x in pitch_track], '.')
plt.show()
```



Refer to the *examples on plotting* for more details on using Parselmouth for plotting.

Importantly, Parselmouth is thus only needed by the server; the client only needs to be able to send a request and read the reply. Consequently, we could even use a different programming language on the client's side. For example, one could make build a HTML page with JavaScript to make the request and do something with the reply:

```
<head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8" />
    <script type="text/javascript" src="jquery.min.js"></script>
    <script type="text/javascript" src="plotly.min.js"></script>
    <script type="text/javascript">
    var update_plot = function() {
        var audio = document.getElementById("audio").files[0];
        var formData = new FormData();
        formData.append("audio", audio);
```

(continues on next page)

(continued from previous page)

```
        $.getJSON({url: "http://127.0.0.1:5000/pitch_track", method: "POST",
                   data: formData, processData: false, contentType: false,
                   success: function(data){
                       Plotly.newPlot("plot", [{ x: [...Array(data.length).keys()],
                                                 y: data.map(function(x) { return x␣
↪== 0.0 ? undefined : x; }),
                                                 type: "lines" }]);}});
    };
    </script>
</head>
<body>
<form onsubmit="update_plot(); return false;">
    <input type="file" name="audio" id="audio" />
    <input type="submit" value="Get pitch track" />
    <div id="plot" style="width:1000px;height:600px;"></div>
</form>
</body>
```

Again, one thing to take into account is the security of running such a web server. However, apart from deploying the flask server in a secure and performant way, we also need one extra thing to circumvent a standard security feature of the browser. Without handling Cross Origin Resource Sharing (CORS) on the server, the JavaScript code on the client side will not be able to access the web service's reply. A Flask extension exists however, Flask-CORS, and we refer to its documentation for further details.

[8]:
```
# Let's shut down the server
p.kill()
```

[9]:
```
# Cleaning up the file that was written to disk
!rm server.py
```

2.6 Projects using Parselmouth

The following projects provide larger, real-life examples and demonstrate the use of Parselmouth:

- The my-voice-analysis and myprosody projects by Shahab Sabahi (@Shahabks) provide Python libraries for voice analysis and acoustical statistics, interfacing Python to his previously developed Praat scripts.
- David R. Feinberg (@drfeinberg) has written multiple Python scripts and programs with Parselmouth to analyse properties of speech recordings:
  - Praat Scripts is a collection of Praat scripts used in research, translated into Python.
  - Voice Lab Software is a GUI application to measure and manipulate voices.

Note: If you have a project using Parselmouth that could be useful for others, and want to add it to this list, do let us know on Gitter!

CHAPTER 3

API Reference

parselmouth.**VERSION = '0.4.0.dev0'**
This version of Parselmouth.

parselmouth.**PRAAT_VERSION = '6.1.15'**
The Praat version on which this version of Parselmouth is based.

parselmouth.**PRAAT_VERSION_DATE = '20 May 2020'**
The release date of the Praat version on which this version of Parselmouth is based.

exception parselmouth.**PraatError**
Bases: RuntimeError

exception parselmouth.**PraatFatal**
Bases: BaseException

exception parselmouth.**PraatWarning**
Bases: UserWarning

class parselmouth.**AmplitudeScaling**
Bases: pybind11_builtins.pybind11_object

Members:

INTEGRAL

SUM

NORMALIZE

PEAK_0_99

__eq__(*self: object*, *arg0: object*) → bool

__hash__(*self: object*) → int_

__init__(**args*, ***kwargs*)
Overloaded function.

1. __init__(self: parselmouth.AmplitudeScaling, arg0: int) -> None

2. __init__(self: parselmouth.AmplitudeScaling, arg0: str) -> None

__int__ (*self: parselmouth.AmplitudeScaling*) → int

__ne__ (*self: object*, *arg0: object*) → bool

__repr__ (*self: handle*) → str

INTEGRAL = AmplitudeScaling.INTEGRAL

NORMALIZE = AmplitudeScaling.NORMALIZE

PEAK_0_99 = AmplitudeScaling.PEAK_0_99

SUM = AmplitudeScaling.SUM

name

class parselmouth.**CC**

Bases: *parselmouth.TimeFrameSampled*, *parselmouth.Sampled*

class Frame

Bases: pybind11_builtins.pybind11_object

__getitem__ (*self: parselmouth.CC.Frame*, *i: int*) → float

__len__ (*self: parselmouth.CC.Frame*) → int

__setitem__ (*self: parselmouth.CC.Frame*, *i: int*, *value: float*) → None

to_array (*self: parselmouth.CC.Frame*) → numpy.ndarray[float64]

__init__

Initialize self. See help(type(self)) for accurate signature.

c

c0

__getitem__ (**args*, ***kwargs*)

Overloaded function.

1. __getitem__(self: parselmouth.CC, i: int) -> parselmouth.CC.Frame

2. __getitem__(self: parselmouth.CC, ij: Tuple[int, int]) -> float

__iter__ (*self: parselmouth.CC*) → iterator

__setitem__ (*self: parselmouth.CC, ij: Tuple[int, int], value: float*) → None

get_c0_value_in_frame (*self: parselmouth.CC, frame_number: Positive[int]*) → float

get_frame (*self: parselmouth.CC, frame_number: Positive[int]*) → parselmouth.CC.Frame

get_number_of_coefficients (*self: parselmouth.CC, frame_number: Positive[int]*) → int

get_value_in_frame (*self: parselmouth.CC, frame_number: Positive[int], index: Positive[int]*) → float

to_array (*self: parselmouth.CC*) → numpy.ndarray[float64]

to_matrix (*self: parselmouth.CC*) → parselmouth.Matrix

__init__

Initialize self. See help(type(self)) for accurate signature.

fmax

fmin

max_n_coefficients

class parselmouth.**Data**

Bases: *parselmouth.Thing*

class FileFormat

Bases: pybind11_builtins.pybind11_object

Members:

TEXT

SHORT_TEXT

BINARY

__eq__(*self: object*, *arg0: object*) → bool

__hash__(*self: object*) → int_

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: parselmouth.Data.FileFormat, arg0: int) -> None
2. __init__(self: parselmouth.Data.FileFormat, arg0: str) -> None

__int__(*self: parselmouth.Data.FileFormat*) → int

__ne__(*self: object*, *arg0: object*) → bool

__repr__(*self: handle*) → str

BINARY = FileFormat.BINARY

SHORT_TEXT = FileFormat.SHORT_TEXT

TEXT = FileFormat.TEXT

name

__copy__(*self: parselmouth.Data*) → parselmouth.Data

__deepcopy__(*self: parselmouth.Data*, *memo: dict*) → parselmouth.Data

__eq__(*self: parselmouth.Data*, *other: parselmouth.Data*) → bool

__ne__(*self: parselmouth.Data*, *other: parselmouth.Data*) → bool

copy(*self: parselmouth.Data*) → parselmouth.Data

static read(*file_path: str*) → parselmouth.Data

Read a file into a *parselmouth.Data* object.

Parameters **file_path** (*str*) – The path of the file on disk to read.

Returns The Praat Data object that was read.

Return type *parselmouth.Data*

See also:

Praat: "Read from file..."

save(*self: parselmouth.Data*, *file_path: str*, *format: parselmouth.Data.FileFormat = FileFormat.TEXT*) → None

save_as_binary_file(*self: parselmouth.Data*, *file_path: str*) → None

save_as_short_text_file(*self: parselmouth.Data*, *file_path: str*) → None

save_as_text_file(*self: parselmouth.Data*, *file_path: str*) → None

__init__
Initialize self. See help(type(self)) for accurate signature.

class parselmouth.**Formant**
Bases: *parselmouth.TimeFrameSampled*, *parselmouth.Sampled*

get_bandwidth_at_time(*self: parselmouth.Formant, formant_number: Positive[int], time: float, unit: parselmouth.FormantUnit = FormantUnit.HERTZ*) → float

get_value_at_time(*self: parselmouth.Formant, formant_number: Positive[int], time: float, unit: parselmouth.FormantUnit = FormantUnit.HERTZ*) → float

__init__
Initialize self. See help(type(self)) for accurate signature.

class parselmouth.**FormantUnit**
Bases: pybind11_builtins.pybind11_object

Members:

HERTZ

BARK

__eq__(*self: object*, *arg0: object*) → bool

__hash__(*self: object*) → int_

__init__(**args*, ***kwargs*)
Overloaded function.

1. __init__(self: parselmouth.FormantUnit, arg0: int) -> None
2. __init__(self: parselmouth.FormantUnit, arg0: str) -> None

__int__(*self: parselmouth.FormantUnit*) → int

__ne__(*self: object*, *arg0: object*) → bool

__repr__(*self: handle*) → str

BARK = FormantUnit.BARK

HERTZ = FormantUnit.HERTZ

name

class parselmouth.**Function**
Bases: *parselmouth.Data*

scale_x_by(*self: parselmouth.Function, scale: Positive[float]*) → None

scale_x_to(*self: parselmouth.Function*, *new_xmin: float*, *new_xmax: float*) → None

shift_x_by(*self: parselmouth.Function*, *shift: float*) → None

shift_x_to(*self: parselmouth.Function*, *x: float*, *new_x: float*) → None

__init__
Initialize self. See help(type(self)) for accurate signature.

xmax

xmin

xrange

class parselmouth.**Harmonicity**
Bases: *parselmouth.TimeFrameSampled*, *parselmouth.Vector*

get_value (*self: parselmouth.Harmonicity*, *time: float*, *interpolation: parselmouth.Interpolation = Interpolation.CUBIC*) → float

__init__
Initialize self. See help(type(self)) for accurate signature.

class parselmouth.**Intensity**
Bases: *parselmouth.TimeFrameSampled*, *parselmouth.Vector*

class AveragingMethod
Bases: pybind11_builtins.pybind11_object

Members:

MEDIAN

ENERGY

SONES

DB

__eq__ (*self: object*, *arg0: object*) → bool

__hash__ (*self: object*) → int_

__init__ (**args*, ***kwargs*)
Overloaded function.
1. __init__(self: parselmouth.Intensity.AveragingMethod, arg0: int) -> None
2. __init__(self: parselmouth.Intensity.AveragingMethod, arg0: str) -> None

__int__ (*self: parselmouth.Intensity.AveragingMethod*) → int

__ne__ (*self: object*, *arg0: object*) → bool

__repr__ (*self: handle*) → str

DB = AveragingMethod.DB

ENERGY = AveragingMethod.ENERGY

MEDIAN = AveragingMethod.MEDIAN

SONES = AveragingMethod.SONES

name

get_average (*self: parselmouth.Intensity*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*, *averaging_method: parselmouth.Intensity.AveragingMethod = AveragingMethod.ENERGY*) → float

get_value (*self: parselmouth.Intensity*, *time: float*, *interpolation: parselmouth.Interpolation = Interpolation.CUBIC*) → float

__init__
Initialize self. See help(type(self)) for accurate signature.

class parselmouth.**Interpolation**
Bases: pybind11_builtins.pybind11_object

Members:

NEAREST

LINEAR

CUBIC

SINC70

SINC700

__eq__ (*self: object*, *arg0: object*) → bool

__hash__ (*self: object*) → int_

__init__ (**args*, ***kwargs*)
Overloaded function.

1. __init__(self: parselmouth.Interpolation, arg0: int) -> None
2. __init__(self: parselmouth.Interpolation, arg0: str) -> None

__int__ (*self: parselmouth.Interpolation*) → int

__ne__ (*self: object*, *arg0: object*) → bool

__repr__ (*self: handle*) → str

CUBIC = Interpolation.CUBIC

LINEAR = Interpolation.LINEAR

NEAREST = Interpolation.NEAREST

SINC70 = Interpolation.SINC70

SINC700 = Interpolation.SINC700

name

class parselmouth.**MFCC**
Bases: *parselmouth.CC*

convolve (*self: parselmouth.MFCC*, *other: parselmouth.MFCC*, *scaling: parselmouth.AmplitudeScaling = AmplitudeScaling.PEAK_0_99*, *signal_outside_time_domain: parselmouth.SignalOutsideTimeDomain = SignalOutsideTimeDomain.ZERO*) → parselmouth.Sound

cross_correlate (*self: parselmouth.MFCC*, *other: parselmouth.MFCC*, *scaling: parselmouth.AmplitudeScaling = AmplitudeScaling.PEAK_0_99*, *signal_outside_time_domain: parselmouth.SignalOutsideTimeDomain = SignalOutsideTimeDomain.ZERO*) → parselmouth.Sound

extract_features (*self: parselmouth.MFCC*, *window_length: Positive[float] = 0.025*, *include_energy: bool = False*) → parselmouth.Matrix

to_matrix_features (*self: parselmouth.MFCC*, *window_length: Positive[float] = 0.025*, *include_energy: bool = False*) → parselmouth.Matrix

to_sound (*self: parselmouth.MFCC*) → parselmouth.Sound

__init__
Initialize self. See help(type(self)) for accurate signature.

class parselmouth.**Matrix**
Bases: *parselmouth.SampledXY*

as_array (*self: parselmouth.Matrix*) → numpy.ndarray[float64]

at_xy (*self: parselmouth.Matrix*, *x: float*, *y: float*) → float

formula (**args*, ***kwargs*)
Overloaded function.

1. formula(self: parselmouth.Matrix, formula: str, from_x: Optional[float] = None, to_x: Optional[float] = None, from_y: Optional[float] = None, to_y: Optional[float] = None) -> None
2. formula(self: parselmouth.Matrix, formula: str, x_range: Tuple[Optional[float], Optional[float]] = (None, None), y_range: Tuple[Optional[float], Optional[float]] = (None, None)) -> None

get_column_distance (*self: parselmouth.Matrix*) → float

get_highest_x (*self: parselmouth.Matrix*) → float

get_highest_y (*self: parselmouth.Matrix*) → float

get_lowest_x (*self: parselmouth.Matrix*) → float

get_lowest_y (*self: parselmouth.Matrix*) → float

get_maximum (*self: parselmouth.Matrix*) → float

get_minimum (*self: parselmouth.Matrix*) → float

get_number_of_columns (*self: parselmouth.Matrix*) → int

get_number_of_rows (*self: parselmouth.Matrix*) → int

get_row_distance (*self: parselmouth.Matrix*) → float

get_sum (*self: parselmouth.Matrix*) → float

get_value_at_xy (*self: parselmouth.Matrix*, *x: float*, *y: float*) → float

get_value_in_cell (*self: parselmouth.Matrix, row_number: Positive[int], column_number: Positive[int]*) → float

get_x_of_column (*self: parselmouth.Matrix, column_number: Positive[int]*) → float

get_y_of_row (*self: parselmouth.Matrix, row_number: Positive[int]*) → float

save_as_headerless_spreadsheet_file (*self: parselmouth.Matrix*, *file_path: str*) → None

save_as_matrix_text_file (*self: parselmouth.Matrix*, *file_path: str*) → None

set_value (*self: parselmouth.Matrix, row_number: Positive[int], column_number: Positive[int], new_value: float*) → None

__init__
Initialize self. See help(type(self)) for accurate signature.

n_columns

n_rows

values

class parselmouth.**Pitch**
Bases: *parselmouth.TimeFrameSampled*, *parselmouth.Sampled*

class Candidate
Bases: pybind11_builtins.pybind11_object

__init__
Initialize self. See help(type(self)) for accurate signature.

frequency

strength

class Frame
Bases: pybind11_builtins.pybind11_object

__getitem__ (*self: parselmouth.Pitch.Frame*, *i: int*) → parselmouth.Pitch.Candidate

__len__ (*self: parselmouth.Pitch.Frame*) → int

as_array (*self: parselmouth.Pitch.Frame*) → array

select (**args*, ***kwargs*)
Overloaded function.
1. select(self: parselmouth.Pitch.Frame, candidate: parselmouth.Pitch.Candidate) -> None
2. select(self: parselmouth.Pitch.Frame, i: int) -> None

unvoice (*self: parselmouth.Pitch.Frame*) → None

__init__
Initialize self. See help(type(self)) for accurate signature.

candidates

intensity

selected

__getitem__ (**args*, ***kwargs*)
Overloaded function.
1. __getitem__(self: parselmouth.Pitch, i: int) -> parselmouth.Pitch.Frame
2. __getitem__(self: parselmouth.Pitch, ij: Tuple[int, int]) -> parselmouth.Pitch.Candidate

__iter__ (*self: parselmouth.Pitch*) → iterator

count_differences (*self: parselmouth.Pitch*, *other: parselmouth.Pitch*) → str

count_voiced_frames (*self: parselmouth.Pitch*) → int

fifth_down (*self: parselmouth.Pitch*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*) → None

fifth_up (*self: parselmouth.Pitch*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*) → None

formula (*self: parselmouth.Pitch*, *formula: str*) → None

get_frame (*self: parselmouth.Pitch, frame_number: Positive[int]*) → parselmouth.Pitch.Frame

get_mean_absolute_slope (*self: parselmouth.Pitch*, *unit: parselmouth.PitchUnit = PitchUnit.HERTZ*) → float

get_slope_without_octave_jumps (*self: parselmouth.Pitch*) → float

get_value_at_time (*self: parselmouth.Pitch*, *time: float*, *unit: parselmouth.PitchUnit = PitchUnit.HERTZ*, *interpolation: parselmouth.Interpolation = Interpolation.LINEAR*) → float

get_value_in_frame (*self: parselmouth.Pitch*, *frame_number: int*, *unit: parselmouth.PitchUnit = PitchUnit.HERTZ*) → float

interpolate (*self: parselmouth.Pitch*) → parselmouth.Pitch

kill_octave_jumps (*self: parselmouth.Pitch*) → parselmouth.Pitch

octave_down (*self: parselmouth.Pitch*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*) → None

octave_up (*self: parselmouth.Pitch*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*) → None

path_finder (*self: parselmouth.Pitch*, *silence_threshold: float = 0.03*, *voicing_threshold: float = 0.45*, *octave_cost: float = 0.01*, *octave_jump_cost: float = 0.35*, *voiced_unvoiced_cost: float = 0.14*, *ceiling: Positive[float] = 600.0*, *pull_formants: bool = False*) → None

smooth (*self: parselmouth.Pitch*, *bandwidth: Positive[float] = 10.0*) → parselmouth.Pitch

step (*self: parselmouth.Pitch*, *step: float*, *precision: Positive[float] = 0.1*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*) → None

subtract_linear_fit (*self: parselmouth.Pitch*, *unit: parselmouth.PitchUnit = PitchUnit.HERTZ*) → parselmouth.Pitch

to_array (*self: parselmouth.Pitch*) → numpy.ndarray[parselmouth.Pitch.Candidate]

to_matrix (*self: parselmouth.Pitch*) → parselmouth.Matrix

to_sound_hum (*self: parselmouth.Pitch*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*) → parselmouth.Sound

to_sound_pulses (*self: parselmouth.Pitch*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*) → parselmouth.Sound

to_sound_sine (*self: parselmouth.Pitch*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*, *sampling_frequency: Positive[float] = 44100.0*, *round_to_nearest_zero_crossing: float = True*) → parselmouth.Sound

unvoice (*self: parselmouth.Pitch*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*) → None

__init__
Initialize self. See help(type(self)) for accurate signature.

ceiling

max_n_candidates

selected

selected_array

class parselmouth.**PitchUnit**
Bases: pybind11_builtins.pybind11_object

Members:

HERTZ

HERTZ_LOGARITHMIC

MEL

LOG_HERTZ

SEMITONES_1

SEMITONES_100

SEMITONES_200

SEMITONES_440

ERB

__eq__ (*self: object*, *arg0: object*) → bool

__hash__ (*self: object*) → int_

__init__ (**args*, ***kwargs*)
Overloaded function.

1. __init__(self: parselmouth.PitchUnit, arg0: int) -> None
2. __init__(self: parselmouth.PitchUnit, arg0: str) -> None

__int__ (*self: parselmouth.PitchUnit*) → int

__ne__ (*self: object*, *arg0: object*) → bool

__repr__ (*self: handle*) → str

ERB = PitchUnit.ERB

HERTZ = PitchUnit.HERTZ

HERTZ_LOGARITHMIC = PitchUnit.HERTZ_LOGARITHMIC

LOG_HERTZ = PitchUnit.LOG_HERTZ

MEL = PitchUnit.MEL

SEMITONES_1 = PitchUnit.SEMITONES_1

SEMITONES_100 = PitchUnit.SEMITONES_100

SEMITONES_200 = PitchUnit.SEMITONES_200

SEMITONES_440 = PitchUnit.SEMITONES_440

name

class parselmouth.**Sampled**

Bases: *parselmouth.Function*

__len__ (*self: parselmouth.Sampled*) → int

x_bins (*self: parselmouth.Sampled*) → numpy.ndarray[float64]

x_grid (*self: parselmouth.Sampled*) → numpy.ndarray[float64]

xs (*self: parselmouth.Sampled*) → numpy.ndarray[float64]

__init__

Initialize self. See help(type(self)) for accurate signature.

dx

nx

x1

class parselmouth.**SampledXY**

Bases: *parselmouth.Sampled*

y_bins (*self: parselmouth.SampledXY*) → numpy.ndarray[float64]

y_grid (*self: parselmouth.SampledXY*) → numpy.ndarray[float64]

ys (*self: parselmouth.SampledXY*) → numpy.ndarray[float64]

__init__

Initialize self. See help(type(self)) for accurate signature.

dy

ny

y1

ymax

ymin

yrange

class parselmouth.**SignalOutsideTimeDomain**

Bases: pybind11_builtins.pybind11_object

Members:

ZERO

SIMILAR

__eq__ (*self: object*, *arg0: object*) → bool

__hash__ (*self: object*) → int_

__init__ (**args*, ***kwargs*)

Overloaded function.

1. __init__(self: parselmouth.SignalOutsideTimeDomain, arg0: int) -> None
2. __init__(self: parselmouth.SignalOutsideTimeDomain, arg0: str) -> None

__int__ (*self: parselmouth.SignalOutsideTimeDomain*) → int

__ne__ (*self: object*, *arg0: object*) → bool

__repr__ (*self: handle*) → str

SIMILAR = SignalOutsideTimeDomain.SIMILAR

ZERO = SignalOutsideTimeDomain.ZERO

name

class parselmouth.**Sound**

Bases: *parselmouth.TimeFrameSampled*, *parselmouth.Vector*

class ToHarmonicityMethod

Bases: pybind11_builtins.pybind11_object

Members:

AC

CC

GNE

__eq__ (*self: object*, *arg0: object*) → bool

__hash__ (*self: object*) → int_

__init__ (**args*, ***kwargs*)

Overloaded function.

1. __init__(self: parselmouth.Sound.ToHarmonicityMethod, arg0: int) -> None
2. __init__(self: parselmouth.Sound.ToHarmonicityMethod, arg0: str) -> None

__int__ (*self: parselmouth.Sound.ToHarmonicityMethod*) → int

__ne__ (*self: object*, *arg0: object*) → bool

__repr__ (*self: handle*) → str

AC = ToHarmonicityMethod.AC

CC = ToHarmonicityMethod.CC

GNE = ToHarmonicityMethod.GNE

name

class ToPitchMethod

Bases: pybind11_builtins.pybind11_object

Members:

AC

CC

SPINET

SHS

__eq__ (*self: object*, *arg0: object*) → bool

__hash__ (*self: object*) → int_

__init__ (**args*, ***kwargs*)

Overloaded function.

1. __init__(self: parselmouth.Sound.ToPitchMethod, arg0: int) -> None
2. __init__(self: parselmouth.Sound.ToPitchMethod, arg0: str) -> None

__int__ (*self: parselmouth.Sound.ToPitchMethod*) → int

__ne__ (*self: object*, *arg0: object*) → bool

__repr__ (*self: handle*) → str

AC = ToPitchMethod.AC

CC = ToPitchMethod.CC

SHS = ToPitchMethod.SHS

SPINET = ToPitchMethod.SPINET

name

__init__ (**args*, ***kwargs*)

Overloaded function.

1. __init__(self: parselmouth.Sound, values: numpy.ndarray[float64], sampling_frequency: Positive[float] = 44100.0, start_time: float = 0.0) -> None
2. __init__(self: parselmouth.Sound, file_path: str) -> None

autocorrelate (*self: parselmouth.Sound*, *scaling: parselmouth.AmplitudeScaling = AmplitudeScaling.PEAK_0_99*, *signal_outside_time_domain: parselmouth.SignalOutsideTimeDomain = SignalOutsideTimeDomain.ZERO*) → parselmouth.Sound

static combine_to_stereo (*sounds: List[parselmouth.Sound]*) → parselmouth.Sound

static concatenate (*sounds: List[parselmouth.Sound], overlap: NonNegative[float] = 0.0*) → parselmouth.Sound

convert_to_mono (*self: parselmouth.Sound*) → parselmouth.Sound

convert_to_stereo (*self: parselmouth.Sound*) → parselmouth.Sound

convolve (*self: parselmouth.Sound*, *other: parselmouth.Sound*, *scaling: parselmouth.AmplitudeScaling = AmplitudeScaling.PEAK_0_99*, *signal_outside_time_domain: parselmouth.SignalOutsideTimeDomain = SignalOutsideTimeDomain.ZERO*) → parselmouth.Sound

cross_correlate (*self: parselmouth.Sound*, *other: parselmouth.Sound*, *scaling: parselmouth.AmplitudeScaling = AmplitudeScaling.PEAK_0_99*, *signal_outside_time_domain: parselmouth.SignalOutsideTimeDomain = SignalOutsideTimeDomain.ZERO*) → parselmouth.Sound

de_emphasize (*self: parselmouth.Sound*, *from_frequency: float = 50.0*, *normalize: bool = True*) → None

deepen_band_modulation (*self: parselmouth.Sound*, *enhancement: Positive[float] = 20.0*, *from_frequency: Positive[float] = 300.0*, *to_frequency: Positive[float] = 8000.0*, *slow_modulation: Positive[float] = 3.0*, *fast_modulation: Positive[float] = 30.0*, *band_smoothing: Positive[float] = 100.0*) → parselmouth.Sound

extract_all_channels (*self: parselmouth.Sound*) → List[parselmouth.Sound]

extract_channel (**args*, ***kwargs*)

Overloaded function.

1. extract_channel(self: parselmouth.Sound, channel: int) -> parselmouth.Sound
2. extract_channel(self: parselmouth.Sound, arg0: str) -> parselmouth.Sound

extract_left_channel (*self: parselmouth.Sound*) → parselmouth.Sound

extract_part (*self: parselmouth.Sound*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*, *window_shape: parselmouth.WindowShape = WindowShape.RECTANGULAR*, *relative_width: Positive[float] = 1.0*, *preserve_times: bool = False*) → parselmouth.Sound

extract_part_for_overlap (*self: parselmouth.Sound, from_time: Optional[float] = None, to_time: Optional[float] = None, overlap: Positive[float]*) → parselmouth.Sound

extract_right_channel (*self: parselmouth.Sound*) → parselmouth.Sound

get_energy (*self: parselmouth.Sound*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*) → float

get_energy_in_air (*self: parselmouth.Sound*) → float

get_index_from_time (*self: parselmouth.Sound*, *time: float*) → float

get_intensity (*self: parselmouth.Sound*) → float

get_nearest_zero_crossing (*self: parselmouth.Sound*, *time: float*, *channel: int = 1*) → float

get_number_of_channels (*self: parselmouth.Sound*) → int

get_number_of_samples (*self: parselmouth.Sound*) → int

get_power (*self: parselmouth.Sound*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*) → float

get_power_in_air (*self: parselmouth.Sound*) → float

get_rms (*self: parselmouth.Sound*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*) → float

get_root_mean_square (*self: parselmouth.Sound*, *from_time: Optional[float] = None*, *to_time: Optional[float] = None*) → float

get_sampling_frequency (*self: parselmouth.Sound*) → float

get_sampling_period (*self: parselmouth.Sound*) → float

get_time_from_index (*self: parselmouth.Sound*, *sample: int*) → float

lengthen (*self: parselmouth.Sound, minimum_pitch: Positive[float] = 75.0, maximum_pitch: Positive[float] = 600.0, factor: Positive[float]*) → parselmouth.Sound

multiply_by_window (*self: parselmouth.Sound, window_shape: parselmouth.WindowShape*) → None

override_sampling_frequency (*self: parselmouth.Sound, new_frequency: Positive[float]*) → None

pre_emphasize (*self: parselmouth.Sound, from_frequency: float = 50.0, normalize: bool = True*) → None

resample (*self: parselmouth.Sound, new_frequency: float, precision: int = 50*) → parselmouth.Sound

reverse (*self: parselmouth.Sound, from_time: Optional[float] = None, to_time: Optional[float] = None*) → None

save (*self: parselmouth.Sound, file_path: str, format: parselmouth.SoundFileFormat*) → None

scale_intensity (*self: parselmouth.Sound, new_average_intensity: float*) → None

set_to_zero (*self: parselmouth.Sound, from_time: Optional[float] = None, to_time: Optional[float] = None, round_to_nearest_zero_crossing: bool = True*) → None

to_formant_burg (*self: parselmouth.Sound, time_step: Optional[Positive[float]] = None, max_number_of_formants: Positive[float] = 5.0, maximum_formant: float = 5500.0, window_length: Positive[float] = 0.025, pre_emphasis_from: Positive[float] = 50.0*) → parselmouth.Formant

to_harmonicity (*self: parselmouth.Sound, method: parselmouth.Sound.ToHarmonicityMethod = ToHarmonicityMethod.CC, *args, **kwargs*) → object

to_harmonicity_ac (*self: parselmouth.Sound, time_step: Positive[float] = 0.01, minimum_pitch: Positive[float] = 75.0, silence_threshold: float = 0.1, periods_per_window: Positive[float] = 1.0*) → parselmouth.Harmonicity

to_harmonicity_cc (*self: parselmouth.Sound, time_step: Positive[float] = 0.01, minimum_pitch: Positive[float] = 75.0, silence_threshold: float = 0.1, periods_per_window: Positive[float] = 1.0*) → parselmouth.Harmonicity

to_harmonicity_gne (*self: parselmouth.Sound, minimum_frequency: Positive[float] = 500.0, maximum_frequency: Positive[float] = 4500.0, bandwidth: Positive[float] = 1000.0, step: Positive[float] = 80.0*) → parselmouth.Matrix

to_intensity (*self: parselmouth.Sound, minimum_pitch: Positive[float] = 100.0, time_step: Optional[Positive[float]] = None, subtract_mean: bool = True*) → parselmouth.Intensity

to_mfcc (*self: parselmouth.Sound, number_of_coefficients: Positive[int] = 12, window_length: Positive[float] = 0.015, time_step: Positive[float] = 0.005, firstFilterFreqency: Positive[float] = 100.0, distance_between_filters: Positive[float] = 100.0, maximum_frequency: Optional[Positive[float]] = None*) → parselmouth.MFCC

to_pitch (**args, **kwargs*)

Overloaded function.

1. to_pitch(self: parselmouth.Sound, time_step: Optional[Positive[float]] = None, pitch_floor: Positive[float] = 75.0, pitch_ceiling: Positive[float] = 600.0) -> parselmouth.Pitch

2. to_pitch(self: parselmouth.Sound, method: parselmouth.Sound.ToPitchMethod, *args, **kwargs) -> object

to_pitch_ac (*self: parselmouth.Sound, time_step: Optional[Positive[float]] = None, pitch_floor: Positive[float] = 75.0, max_number_of_candidates: Positive[int] = 15, very_accurate: bool = False, silence_threshold: float = 0.03, voicing_threshold: float = 0.45, octave_cost: float = 0.01, octave_jump_cost: float = 0.35, voiced_unvoiced_cost: float = 0.14, pitch_ceiling: Positive[float] = 600.0*) → parselmouth.Pitch

to_pitch_cc(*self: parselmouth.Sound*, *time_step: Optional[Positive[float]] = None*, *pitch_floor: Positive[float] = 75.0*, *max_number_of_candidates: Positive[int] = 15*, *very_accurate: bool = False*, *silence_threshold: float = 0.03*, *voicing_threshold: float = 0.45*, *octave_cost: float = 0.01*, *octave_jump_cost: float = 0.35*, *voiced_unvoiced_cost: float = 0.14*, *pitch_ceiling: Positive[float] = 600.0*) → parselmouth.Pitch

to_pitch_shs(*self: parselmouth.Sound*, *time_step: Positive[float] = 0.01*, *minimum_pitch: Positive[float] = 50.0*, *max_number_of_candidates: Positive[int] = 15*, *maximum_frequency_component: Positive[float] = 1250.0*, *max_number_of_subharmonics: Positive[int] = 15*, *compression_factor: Positive[float] = 0.84*, *ceiling: Positive[float] = 600.0*, *number_of_points_per_octave: Positive[int] = 48*) → parselmouth.Pitch

to_pitch_spinet(*self: parselmouth.Sound*, *time_step: Positive[float] = 0.005*, *window_length: Positive[float] = 0.04*, *minimum_filter_frequency: Positive[float] = 70.0*, *maximum_filter_frequency: Positive[float] = 5000.0*, *number_of_filters: Positive[int] = 250*, *ceiling: Positive[float] = 500.0*, *max_number_of_candidates: Positive[int] = 15*) → parselmouth.Pitch

to_spectrogram(*self: parselmouth.Sound*, *window_length: Positive[float] = 0.005*, *maximum_frequency: Positive[float] = 5000.0*, *time_step: Positive[float] = 0.002*, *frequency_step: Positive[float] = 20.0*, *window_shape: parselmouth.SpectralAnalysisWindowShape = SpectralAnalysisWindowShape.GAUSSIAN*) → parselmouth.Spectrogram

to_spectrum(*self: parselmouth.Sound*, *fast: bool = True*) → parselmouth.Spectrum

n_channels

n_samples

sampling_frequency

sampling_period

class parselmouth.**SoundFileFormat**

Bases: pybind11_builtins.pybind11_object

Members:

WAV

AIFF

AIFC

NEXT_SUN

NIST

FLAC

KAY

SESAM

WAV_24

WAV_32

RAW_8_SIGNED

RAW_8_UNSIGNED

RAW_16_BE

RAW_16_LE

RAW_24_BE

RAW_24_LE

RAW_32_BE

RAW_32_LE

__eq__ (*self: object*, *arg0: object*) → bool

__hash__ (*self: object*) → int_

__init__ (**args*, ***kwargs*)
Overloaded function.

1. __init__(self: parselmouth.SoundFileFormat, arg0: int) -> None
2. __init__(self: parselmouth.SoundFileFormat, arg0: str) -> None

__int__ (*self: parselmouth.SoundFileFormat*) → int

__ne__ (*self: object*, *arg0: object*) → bool

__repr__ (*self: handle*) → str

AIFC = SoundFileFormat.AIFC

AIFF = SoundFileFormat.AIFF

FLAC = SoundFileFormat.FLAC

KAY = SoundFileFormat.KAY

NEXT_SUN = SoundFileFormat.NEXT_SUN

NIST = SoundFileFormat.NIST

RAW_16_BE = SoundFileFormat.RAW_16_BE

RAW_16_LE = SoundFileFormat.RAW_16_LE

RAW_24_BE = SoundFileFormat.RAW_24_BE

RAW_24_LE = SoundFileFormat.RAW_24_LE

RAW_32_BE = SoundFileFormat.RAW_32_BE

RAW_32_LE = SoundFileFormat.RAW_32_LE

RAW_8_SIGNED = SoundFileFormat.RAW_8_SIGNED

RAW_8_UNSIGNED = SoundFileFormat.RAW_8_UNSIGNED

SESAM = SoundFileFormat.SESAM

WAV = SoundFileFormat.WAV

WAV_24 = SoundFileFormat.WAV_24

WAV_32 = SoundFileFormat.WAV_32

name

class parselmouth.**SpectralAnalysisWindowShape**
Bases: pybind11_builtins.pybind11_object

Members:

SQUARE

HAMMING

BARTLETT

WELCH

HANNING

GAUSSIAN

__eq__ (*self: object*, *arg0: object*) → bool

__hash__ (*self: object*) → int_

__init__ (**args*, ***kwargs*)
Overloaded function.

1. __init__(self: parselmouth.SpectralAnalysisWindowShape, arg0: int) -> None
2. __init__(self: parselmouth.SpectralAnalysisWindowShape, arg0: str) -> None

__int__ (*self: parselmouth.SpectralAnalysisWindowShape*) → int

__ne__ (*self: object*, *arg0: object*) → bool

__repr__ (*self: handle*) → str

BARTLETT = SpectralAnalysisWindowShape.BARTLETT

GAUSSIAN = SpectralAnalysisWindowShape.GAUSSIAN

HAMMING = SpectralAnalysisWindowShape.HAMMING

HANNING = SpectralAnalysisWindowShape.HANNING

SQUARE = SpectralAnalysisWindowShape.SQUARE

WELCH = SpectralAnalysisWindowShape.WELCH

name

class parselmouth.**Spectrogram**
Bases: *parselmouth.TimeFrameSampled*, *parselmouth.Matrix*

get_power_at (*self: parselmouth.Spectrogram*, *time: float*, *frequency: float*) → float

synthesize_sound (*self: parselmouth.Spectrogram*, *sampling_frequency: Positive[float] = 44100.0*) → parselmouth.Sound

to_sound (*self: parselmouth.Spectrogram*, *sampling_frequency: Positive[float] = 44100.0*) → parselmouth.Sound

to_spectrum_slice (*self: parselmouth.Spectrogram*, *time: float*) → parselmouth.Spectrum

__init__
Initialize self. See help(type(self)) for accurate signature.

class parselmouth.**Spectrum**
Bases: *parselmouth.Matrix*

__getitem__ (*self: parselmouth.Spectrum*, *index: int*) → complex

__init__ (**args*, ***kwargs*)
Overloaded function.

1. __init__(self: parselmouth.Spectrum, values: numpy.ndarray[float64], maximum_frequency: Positive[float]) -> None
2. __init__(self: parselmouth.Spectrum, values: numpy.ndarray[complex128], maximum_frequency: Positive[float]) -> None

__setitem__(*self: parselmouth.Spectrum*, *index: int*, *value: complex*) → None

cepstral_smoothing(*self: parselmouth.Spectrum*, *bandwidth: Positive[float] = 500.0*) → parselmouth.Spectrum

get_band_density(**args*, ***kwargs*)
Overloaded function.

1. get_band_density(self: parselmouth.Spectrum, band_floor: Optional[float] = None, band_ceiling: Optional[float] = None) -> float

2. get_band_density(self: parselmouth.Spectrum, band: Tuple[Optional[float], Optional[float]] = (None, None)) -> float

get_band_density_difference(**args*, ***kwargs*)
Overloaded function.

1. get_band_density_difference(self: parselmouth.Spectrum, low_band_floor: Optional[float] = None, low_band_ceiling: Optional[float] = None, high_band_floor: Optional[float] = None, high_band_ceiling: Optional[float] = None) -> float

2. get_band_density_difference(self: parselmouth.Spectrum, low_band: Tuple[Optional[float], Optional[float]] = (None, None), high_band: Tuple[Optional[float], Optional[float]] = (None, None)) -> float

get_band_energy(**args*, ***kwargs*)
Overloaded function.

1. get_band_energy(self: parselmouth.Spectrum, band_floor: Optional[float] = None, band_ceiling: Optional[float] = None) -> float

2. get_band_energy(self: parselmouth.Spectrum, band: Tuple[Optional[float], Optional[float]] = (None, None)) -> float

get_band_energy_difference(**args*, ***kwargs*)
Overloaded function.

1. get_band_energy_difference(self: parselmouth.Spectrum, low_band_floor: Optional[float] = None, low_band_ceiling: Optional[float] = None, high_band_floor: Optional[float] = None, high_band_ceiling: Optional[float] = None) -> float

2. get_band_energy_difference(self: parselmouth.Spectrum, low_band: Tuple[Optional[float], Optional[float]] = (None, None), high_band: Tuple[Optional[float], Optional[float]] = (None, None)) -> float

get_bin_number_from_frequency(*self: parselmouth.Spectrum*, *frequency: float*) → float

get_bin_width(*self: parselmouth.Spectrum*) → float

get_center_of_gravity(*self: parselmouth.Spectrum*, *power: Positive[float] = 2.0*) → float

get_central_moment(*self: parselmouth.Spectrum, moment: Positive[float], power: Positive[float] = 2.0*) → float

get_centre_of_gravity(*self: parselmouth.Spectrum*, *power: Positive[float] = 2.0*) → float

get_frequency_from_bin_number(*self: parselmouth.Spectrum, band_number: Positive[int]*) → float

get_highest_frequency(*self: parselmouth.Spectrum*) → float

get_imaginary_value_in_bin(**args*, ***kwargs*)
Overloaded function.

1. get_imaginary_value_in_bin(self: parselmouth.Spectrum, bin_number: Positive[int]) -> float

2. get_imaginary_value_in_bin(self: parselmouth.Spectrum, bin_number: Positive[int], value: float) -> None

get_kurtosis (*self: parselmouth.Spectrum*, *power: Positive[float] = 2.0*) → float

get_lowest_frequency (*self: parselmouth.Spectrum*) → float

get_number_of_bins (*self: parselmouth.Spectrum*) → int

get_real_value_in_bin (*self: parselmouth.Spectrum, bin_number: Positive[int]*) → float

get_skewness (*self: parselmouth.Spectrum*, *power: Positive[float] = 2.0*) → float

get_standard_deviation (*self: parselmouth.Spectrum*, *power: Positive[float] = 2.0*) → float

get_value_in_bin (**args*, ***kwargs*)
Overloaded function.

1. get_value_in_bin(self: parselmouth.Spectrum, bin_number: Positive[int]) -> complex

2. get_value_in_bin(self: parselmouth.Spectrum, bin_number: Positive[int], value: complex) -> None

lpc_smoothing (*self: parselmouth.Spectrum*, *num_peaks: Positive[int] = 5*, *pre_emphasis_from: Positive[float] = 50.0*) → parselmouth.Spectrum

set_real_value_in_bin (*self: parselmouth.Spectrum, bin_number: Positive[int], value: float*) → None

to_sound (*self: parselmouth.Spectrum*) → parselmouth.Sound

to_spectrogram (*self: parselmouth.Spectrum*) → parselmouth.Spectrogram

bin_width

df

fmax

fmin

highest_frequency

lowest_frequency

n_bins

nf

class parselmouth.**TextGrid**
Bases: *parselmouth.Function*

__init__ (**args*, ***kwargs*)
Overloaded function.

1. __init__(self: parselmouth.TextGrid, start_time: float, end_time: float, tier_names: str, point_tier_names: str) -> None

2. __init__(self: parselmouth.TextGrid, start_time: float, end_time: float, tier_names: List[str] = [], point_tier_names: List[str] = []) -> None

3. __init__(self: parselmouth.TextGrid, tgt_text_grid: tgt.core.TextGrid) -> None

static from_tgt (*tgt_text_grid: tgt.core.TextGrid*) → parselmouth.TextGrid

to_tgt (*self: parselmouth.TextGrid*) → tgt.core.TextGrid

class parselmouth.**Thing**
Bases: pybind11_builtins.pybind11_object

__str__ (*self: parselmouth.Thing*) → str

info (*self: parselmouth.Thing*) → str

__init__
Initialize self. See help(type(self)) for accurate signature.

class_name

full_name

name

class parselmouth.**TimeFrameSampled**
Bases: *parselmouth.TimeFunction*, *parselmouth.Sampled*

frame_number_to_time (*self: parselmouth.Sampled, frame_number: Positive[int]*) → float

get_frame_number_from_time (*self: parselmouth.Sampled*, *time: float*) → float

get_number_of_frames (*self: parselmouth.Sampled*) → int

get_time_from_frame_number (*self: parselmouth.Sampled, frame_number: Positive[int]*) → float

get_time_step (*self: parselmouth.Sampled*) → float

t_bins (*self: parselmouth.Sampled*) → numpy.ndarray[float64]

t_grid (*self: parselmouth.Sampled*) → numpy.ndarray[float64]

time_to_frame_number (*self: parselmouth.Sampled*, *time: float*) → float

ts (*self: parselmouth.Sampled*) → numpy.ndarray[float64]

__init__
Initialize self. See help(type(self)) for accurate signature.

dt

n_frames

nt

t1

time_step

class parselmouth.**TimeFunction**
Bases: *parselmouth.Function*

get_end_time (*self: parselmouth.Function*) → float

get_start_time (*self: parselmouth.Function*) → float

get_total_duration (*self: parselmouth.Function*) → float

scale_times_by (*self: parselmouth.Function, scale: Positive[float]*) → None

scale_times_to (*self: parselmouth.Function*, *new_start_time: float*, *new_end_time: float*) → None

shift_times_by (*self: parselmouth.Function*, *seconds: float*) → None

shift_times_to (**args*, ***kwargs*)
Overloaded function.

1. shift_times_to(self: parselmouth.Function, time: float, new_time: float) -> None
2. shift_times_to(self: parselmouth.Function, time: str, new_time: float) -> None

__init__
Initialize self. See help(type(self)) for accurate signature.

centre_time

duration

end_time

start_time

time_range

tmax

tmin

total_duration

trange

class parselmouth.**Vector**
Bases: *parselmouth.Matrix*

__add__ (*self: parselmouth.Vector*, *number: float*) → parselmouth.Vector

__iadd__ (*self: parselmouth.Vector*, *number: float*) → parselmouth.Vector

__imul__ (*self: parselmouth.Vector*, *factor: float*) → parselmouth.Vector

__isub__ (*self: parselmouth.Vector*, *number: float*) → parselmouth.Vector

__itruediv__ (*self: parselmouth.Vector*, *factor: float*) → parselmouth.Vector

__mul__ (*self: parselmouth.Vector*, *factor: float*) → parselmouth.Vector

__radd__ (*self: parselmouth.Vector*, *number: float*) → parselmouth.Vector

__rmul__ (*self: parselmouth.Vector*, *factor: float*) → parselmouth.Vector

__sub__ (*self: parselmouth.Vector*, *number: float*) → parselmouth.Vector

__truediv__ (*self: parselmouth.Vector*, *factor: float*) → parselmouth.Vector

add (*self: parselmouth.Vector*, *number: float*) → None

divide (*self: parselmouth.Vector*, *factor: float*) → None

get_value (*self: parselmouth.Vector*, *x: float*, *channel: Optional[int] = None*, *interpolation: parselmouth.Interpolation = Interpolation.CUBIC*) → float

multiply (*self: parselmouth.Vector*, *factor: float*) → None

scale (*self: parselmouth.Vector, scale: Positive[float]*) → None

scale_peak (*self: parselmouth.Vector*, *new_peak: Positive[float] = 0.99*) → None

subtract (*self: parselmouth.Vector*, *number: float*) → None

subtract_mean (*self: parselmouth.Vector*) → None

__init__
Initialize self. See help(type(self)) for accurate signature.

class parselmouth.**WindowShape**
Bases: pybind11_builtins.pybind11_object

Members:

RECTANGULAR

TRIANGULAR

PARABOLIC

HANNING

HAMMING

GAUSSIAN1

GAUSSIAN2

GAUSSIAN3

GAUSSIAN4

GAUSSIAN5

KAISER1

KAISER2

__eq__ (*self: object*, *arg0: object*) → bool

__hash__ (*self: object*) → int_

__init__ (**args*, ***kwargs*)
Overloaded function.

1. __init__(self: parselmouth.WindowShape, arg0: int) -> None
2. __init__(self: parselmouth.WindowShape, arg0: str) -> None

__int__ (*self: parselmouth.WindowShape*) → int

__ne__ (*self: object*, *arg0: object*) → bool

__repr__ (*self: handle*) → str

GAUSSIAN1 = WindowShape.GAUSSIAN1

GAUSSIAN2 = WindowShape.GAUSSIAN2

GAUSSIAN3 = WindowShape.GAUSSIAN3

GAUSSIAN4 = WindowShape.GAUSSIAN4

GAUSSIAN5 = WindowShape.GAUSSIAN5

HAMMING = WindowShape.HAMMING

HANNING = WindowShape.HANNING

KAISER1 = WindowShape.KAISER1

KAISER2 = WindowShape.KAISER2

PARABOLIC = WindowShape.PARABOLIC

RECTANGULAR = WindowShape.RECTANGULAR

TRIANGULAR = WindowShape.TRIANGULAR

name

parselmouth.**read**(*file_path: str*) → parselmouth.Data
Read a file into a *parselmouth.Data* object.

Parameters **file_path** (*str*) – The path of the file on disk to read.

Returns The Praat Data object that was read.

Return type *parselmouth.Data*

See also:

Praat: "Read from file…"

parselmouth.praat.**call**(**args*, ***kwargs*)

Overloaded function.

1. call(command: str, *args, **kwargs) -> object
2. call(object: parselmouth.Data, command: str, *args, **kwargs) -> object
3. call(objects: List[parselmouth.Data], command: str, *args, **kwargs) -> object

Call a Praat command.

This function provides a Python interface to call available Praat commands based on the label in the Praat user interface and documentation, similar to the Praat scripting language.

Calling a Praat command through this function roughly corresponds to the following scenario in the Praat user interface or scripting language:

1. Zero, one, or multiple *parselmouth.Data* objects are put into Praat's global object list and are 'selected'.
2. The Python argument values are converted into Praat values; see below.
3. The Praat command is executed on the selected objects with the converted values as arguments.
4. The result of the command is returned. The type of the result depends on the result of the Praat command; see below.
5. Praat's object list is emptied again, such that a future execution of this function is independent from the current call.

The use of *call* is demonstrated in the *Pitch manipulation and Praat commands* example.

Parameters

- **object** (parselmouth.Data) – A single object to add to the Praat object list, which will be selected when the Praat command is called.
- **objects** (*List*[parselmouth.Data]) – Multiple objects to be added to the Praat object list, which will be selected when the Praat command is called.
- **command** (*str*) – The Praat action to call. This is the same command name as one would use in a Praat script and corresponds to the label on the button in the Praat user interface.
- ***args** – The list of values to be passed as arguments to the Praat command. Allowed types for these arguments are:
  - int or float: passed as a Praat numeric value
  - bool: converted into "yes"/"no"
  - str: passed as Praat string value
  - numpy.ndarray: passed as Praat vector or matrix, if the array contains numeric values and is 1D or 2D, respectively.

Keyword Arguments

- **extra_objects** (*List*[parselmouth.Data]) – Extra objects added to the Praat object list that will not be selected when the command is called (default value: []).

- **return_string** (*bool*) – Return the raw string written in the Praat info window instead of the converted Python object (default value: `False`).

Returns

The result of the Praat command. The actual value returned depends on what the Praat command does. The following types can be returned:

- If `return_string=True` was passed, a `str` value is returned, which contains the text that would have been written to the Praat info window.
- A `float`, `int`, `bool`, or `complex` value is returned when the Praat command would write such a value to the Praat info window.
- A `numpy.ndarray` value is returned if the command returns a Praat vector or matrix.
- A *`parselmouth.Data`* object is returned if the command always creates exactly one object. If the actual type of the Praat object is available in Parselmouth, an object of a subtype of *`parselmouth.Data`* is returned.
- A list of *`parselmouth.Data`* objects is returned if the command can create multiple new objects (even if this particular execution of the command only added one object to the Praat object list).
- A `str` is returned when a string or info text would be written to the Praat info window.

Return type object

See also:

`parselmouth.praat.run()`, *`parselmouth.praat.run_file()`*, *Praat: "Scripting"*

parselmouth.praat.**run**(**args*, ***kwargs*)

Overloaded function.

1. run(script: str, *args, **kwargs) -> object
2. run(object: parselmouth.Data, script: str, *args, **kwargs) -> object
3. run(objects: List[parselmouth.Data], script: str, *args, **kwargs) -> object

Run a Praat script.

Given a string with the contents of a Praat script, run this script as if it was run inside Praat itself. Similarly to *`parselmouth.praat.call`*, Parselmouth objects and Python argument values can be passed into the script.

Calling this function roughly corresponds to the following sequence of steps in Praat:

1. Zero, one, or multiple *`parselmouth.Data`* objects are put into Praat's global object list and are 'selected'.
2. The Python argument values are converted into Praat values; see *`call`*.
3. The Praat script is opened and run with the converted values as arguments; see *Praat: "Scripting 6.1. Arguments to the script"*.
4. The results of the execution of the script are returned; see below.
5. Praat's object list is emptied again, such that a future execution of this function is independent from the current call.

Note that the script will be run in Praat's so-called 'batch' mode; see *Praat: "Scripting 6.9. Calling from the command line"*. Since the script is run from inside a Python program, the Praat functionality is run without graphical user interface and no windows (such as *"View & Edit"*) can be opened by the Praat script. However, the functionality in these windows is also available in different ways: for example, opening a *Sound* object in a

"View & Edit" window, making a selection, and choosing *"Extract selected sound (windowed)..."* can also be achieved by directly using the *"Extract part..."* command of the *Sound* object.

Parameters

- **object** (parselmouth.Data) – A single object to add to the Praat object list, which will be selected when the Praat script is run.
- **objects** (*List[*parselmouth.Data*]*) – Multiple objects to be added to the Praat object list, which will be selected when the Praat script is run.
- **script** (*str*) – The content of the Praat script to be run.
- ***args** – The list of values to be passed as arguments to the Praat script. For more details on the allowed types of these argument, see *call*.

Keyword Arguments

- **extra_objects** (*List[*parselmouth.Data*]*) – Extra objects added to the Praat object list that will not be selected when the command is called (default value: `[]`).
- **capture_output** (*bool*) – Intercept and also return the output written to the Praat info window, instead of forwarding it to the Python standard output; see below (default value: `False`).
- **return_variables** (*bool*) – Also return a `dict` of the Praat variables and their values at the end of the script's execution; see below (default value: `False`).

Returns

A list of *parselmouth.Data* objects selected at the end of the script's execution.

Optionally, extra values are returned:

- A `str` containing the intercepted output if `capture_output=True` was passed.
- A `dict` mapping variable names (`str`) to their values (`object`) if `return_variables` is `True`. The values of Praat's variables get converted to Python values:
  - A Praat string variable, with a name ending in `$`, is returned as `str` value.
  - A Praat vector or matrix variable, respectively ending in `#` or `##`, is returned as `numpy.ndarray`.
  - A numeric variable, without variable name suffix, is converted to a Python `float`.

Return type object

See also:

parselmouth.praat.run_file(), *parselmouth.praat.call()*, *Praat: "Scripting"*

parselmouth.praat.**run_file**(**args*, ***kwargs*)

Overloaded function.

1. run_file(path: str, *args, **kwargs) -> object
2. run_file(object: parselmouth.Data, path: str, *args, **kwargs) -> object
3. run_file(objects: List[parselmouth.Data], path: str, *args, **kwargs) -> object

Run a Praat script from file.

Given the filename of a Praat script, the script is read and run the same way as a script string passed to *parselmouth.praat.run*. See *run* for details on the manner in which the script gets executed.

One thing to note is that relative filenames in the Praat script (including those in potential 'include' statements in the script; see *Praat: "Scripting 5.8. Including other scripts"*) will be resolved relative to the path of the script file, just like in Praat. Also note that Praat accomplishes this by temporarily changing the current working during the execution of the script.

Parameters

- **object** (parselmouth.Data) – A single object to add to the Praat object list, which will be selected when the Praat script is run.
- **objects** (*List*[parselmouth.Data]) – Multiple objects to be added to the Praat object list, which will be selected when the Praat script is run.
- **path** (*str*) – The filename of the Praat script to run.
- ***args** – The list of values to be passed as arguments to the Praat script. For more details on the allowed types of these argument, see *call*.

Keyword Arguments

- **keep_cwd** (*bool*) – Keep the current working directory (see os.getcwd) when running the script, rather than changing it to the script's parent directory, as Praat does by default (default value: False). Note that even when set to True, the filenames in the Praat script's include statements will be resolved relatively to the directory containing the script.
- ****kwargs** – See *parselmouth.praat.run*.

Returns See *parselmouth.praat.run*.

Return type object

See also:

parselmouth.praat.run(), *parselmouth.praat.call()*, *Praat: "Scripting"*

CHAPTER 4

Citing Parselmouth

A manuscript introducing Parselmouth (and supplementary material) has been published in the Journal of Phonetics. Scientific work and publications can for now cite Parselmouth in the following way:

> Jadoul, Y., Thompson, B., & de Boer, B. (2018). Introducing Parselmouth: A Python interface to Praat. *Journal of Phonetics*, *71*, 1-15. https://doi.org/10.1016/j.wocn.2018.07.001

```
@article{parselmouth,
    author = "Yannick Jadoul and Bill Thompson and Bart de Boer",
    title = "Introducing {P}arselmouth: A {P}ython interface to {P}raat",
    journal = "Journal of Phonetics",
    volume = "71",
    pages = "1--15",
    year = "2018",
    doi = "https://doi.org/10.1016/j.wocn.2018.07.001"
}
```

Since Parselmouth exposes existing Praat functionality and algorithm implementations, we suggest also citing Praat when using Parselmouth in scientific research:

> Boersma, P., & Weenink, D. (2018). Praat: doing phonetics by computer [Computer program]. Version 6.0.43, retrieved 8 September 2018 from http://www.praat.org/

```
@misc{praat,
    author = "Paul Boersma and David Weenink",
    title = "{P}raat: doing phonetics by computer [{C}omputer program]",
    howpublished = "Version 6.0.43, retrieved 8 September 2018 \url{http://www.praat.
↪org/}",
    year = "2018"
}
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z