Controller listeners

These objects can be used to create MIDI and/or OSC listeners without the need to boot ands start an audio server before receiving messages.

MidiListener

class **`MidiListener`**(*function*, *mididev=-1*, *reportdevice=False*) [source]

Self-contained midi listener thread.

This object allows to setup a Midi server that is independent of the audio server (mainly to be able to receive Midi data even when the audio server is stopped). Although it runs in a separated thread, the same device can’t be used by this object and the audio server at the same time. It is adviced to call the deactivateMidi() method on the audio server to avoid conflicts.

Parent: threading.Thread

Args: function: Python function (can’t be a list)

Function that will be called when a new midi event is available. This function is called with the incoming midi data as arguments. The signature of the function must be:
def myfunc(status, data1, data2)

mididev: int or list of ints, optional

Sets the midi input device (see *pm_list_devices()* for the available devices). The default, -1, means the system default device. A number greater than the highest portmidi device index will open all available input devices. Specific devices can be set with a list of integers.

reportdevice: boolean, optional

If True, the device ID will be reported as a fourth argument to the callback. The signature will then be:
def myfunc(status, data1, data2, id)
Available at initialization only. Defaults to False.

Note: This object is available only if pyo is built with portmidi support (see withPortmidi function).

```
>>> s = Server()
>>> s.deactivateMidi()
>>> s.boot()
>>> def midicall(status, data1, data2):
...     print(status, data1, data2)
>>> listen = MidiListener(midicall, 5)
>>> listen.start()
```

`run`() [source]

Starts the process. The thread runs as daemon, so no need to stop it.

stop() [source]

Stops the listener and properly close the midi ports.

getDeviceInfos() [source]

Returns infos about connected midi devices.

This method returns a list of dictionaries, one per device.

Dictionary format is:

{“id”: device_id (int), “name”: device_name (str), “interface”: interface (str)}

MidiDispatcher

class **MidiDispatcher**(*mididev=-1*) [source]

Self-contained midi dispatcher thread.

This object allows to setup a Midi server that is independent of the audio server (mainly to be able to send Midi data even when the audio server is stopped). Although it runs in a separated thread, the same device can’t be used by this object and the audio server at the same time. It is adviced to call the deactivateMidi() method on the audio server to avoid conflicts.

Use the *send* method to send midi event to connected devices.

Use the *sendx* method to send sysex event to connected devices.

Parent: threading.Thread

Args: mididev: int or list of ints, optional

Sets the midi output device (see *pm_list_devices()* for the available devices). The default, -1, means the system default device. A number greater than the highest portmidi device index will open all available input devices. Specific devices can be set with a list of integers.

Note: This object is available only if pyo is built with portmidi support (see withPortmidi function).

```
>>> s = Server()
>>> s.deactivateMidi()
>>> s.boot()
>>> dispatch = MidiDispatcher(5)
>>> dispatch.start()
>>> dispatch.send(144, 60, 127)
```

run() [source]

Starts the process. The thread runs as daemon, so no need to stop it.

send(*status*, *data1*, *data2=0*, *timestamp=0*, *device=-1*) [source]

Send a MIDI message to the selected midi output device.

Arguments can be list of values to generate multiple events in one call.

Args: status: int

Status byte.

data1: int

First data byte.

data2: int, optional

Second data byte. Defaults to 0.

timestamp: int, optional

The delay time, in milliseconds, before the note is sent on the portmidi stream. A value of 0 means to play the note now. Defaults to 0.

device: int, optional

The index of the device to which the message will be sent. The default (-1) means all devices. See *getDeviceInfos()* to retrieve device indexes.

`sendx`(*msg*, *timestamp=0*, *device=-1*) [source]

Send a MIDI system exclusive message to the selected midi output device.

Arguments can be list of values to generate multiple events in one call.

Args: msg: str

A valid system exclusive message as a string. The first byte must be 0xf0 and the last one must be 0xf7.

timestamp: int, optional

The delay time, in milliseconds, before the note is sent on the portmidi stream. A value of 0 means to play the note now. Defaults to 0.

device: int, optional

The index of the device to which the message will be sent. The default (-1) means all devices. See *getDeviceInfos()* to retrieve device indexes.

`getDeviceInfos`() [source]

Returns infos about connected midi devices.

This method returns a list of dictionaries, one per device.

Dictionary format is:

{“id”: device_id (int), “name”: device_name (str), “interface”: interface (str)}

OscListener

class **`OscListener`**(*function*, *port=9000*) [source]

Self-contained OSC listener thread.

This object allows to setup an OSC server that is independent of the audio server (mainly to be able to receive OSC data even when the audio server is stopped).

Parent: threadind.Thread

Args: function: Python function (can’t be a list)

> Function that will be called when a new OSC event is available. This function is called with the incoming address and values as arguments. The signature of the function must be:
> def myfunc(address, *args)

port: int, optional

> The OSC port on which the values are received. Defaults to 9000.

```
>>> s = Server().boot()
>>> def call(address, *args):
...     print(address, args)
>>> listen = OscListener(call, 9901)
>>> listen.start()
```

run() [source]

Starts the process. The thread runs as daemon, so no need to stop it.