python-osc Documentation

Release 1.7.1

attwad

Aug 05, 2023

Contents:

CHAPTER 1

Dispatcher

The dispatcher maps OSC addresses to functions and calls the functions with the messages' arguments. Function can also be mapped to wildcard addresses.

1.1 Example

```
from pythonosc.dispatcher import Dispatcher
from typing import List, Any

dispatcher = Dispatcher()


def set_filter(address: str, *args: List[Any]) -> None:
    # We expect two float arguments
    if not len(args) == 2 or type(args[0]) is not float or type(args[1]) is not float:
        return

    # Check that address starts with filter
    if not address[:-1] == "/filter":  # Cut off the last character
        return

    value1 = args[0]
    value2 = args[1]
    filterno = address[-1]
    print(f"Setting filter {filterno} values: {value1}, {value2}")


dispatcher.map("/filter*", set_filter)  # Map wildcard address to set_filter function

# Set up server and client for testing
from pythonosc.osc_server import BlockingOSCUDPServer
from pythonosc.udp_client import SimpleUDPClient
```

(continues on next page)

(continued from previous page)

```
server = BlockingOSCUDPServer(("127.0.0.1", 1337), dispatcher)
client = SimpleUDPClient("127.0.0.1", 1337)

# Send message and receive exactly one message (blocking)
client.send_message("/filter1", [1., 2.])
server.handle_request()

client.send_message("/filter8", [6., -2.])
server.handle_request()
```

1.2 Mapping

The dispatcher associates addresses with functions by storing them in a mapping. An address can contains wildcards as defined in the OSC specifications. Call the `Dispatcher.map` method with an address pattern and a handler callback function:

```
from pythonosc.dispatcher import Dispatcher
disp = Dispatcher()
disp.map("/some/address*", some_printing_func)
```

This will for example print any OSC messages starting with `/some/address`.

Additionally you can provide any amount of extra fixed argument that will always be passed before the OSC message arguments:

```
handler = disp.map("/some/other/address", some_printing_func, "This is a fixed arg",
↪"and this is another fixed arg")
```

The handler callback signature must look like this:

```
def some_callback(address: str, *osc_arguments: List[Any]) -> None:
def some_callback(address: str, fixed_argument: List[Any], *osc_arguments: List[Any])␣
↪-> None:
```

Instead of a list you can of course also use a fixed amount of arguments for `osc_arguments`

The `Dispatcher.map` method returns a `Handler` object, which can be used to remove the mapping from the dispatcher.

1.3 Unmapping

A mapping can be undone with the `Dispatcher.unmap` method, which takes an address and `Handler` object as arguments. For example, to unmap the mapping from the *Mapping* section:

```
disp.unmap("some/other/address", handler)
```

Alternatively the handler can be reconstructed from a function and optional fixed argument:

```
disp.unmap("some/other/address", some_printing_func, *some_fixed_args)
```

If the provided mapping doesn't exist, a `ValueError` is raised.

1.4 Default Handler

It is possible to specify a handler callback function that is called for every unmatched address:

```
disp.set_default_handler(some_handler_function)
```

This is extremely useful if you quickly need to find out what addresses an undocumented device is transmitting on or for building a learning function for some controls. The handler must have the same signature as map callbacks:

```
def some_callback(address: str, *osc_arguments: List[Any]) -> None:
```

1.5 Dispatcher Module Documentation

CHAPTER 2

Client

The client allows you to connect and send messages to an OSC server. The client class expects an `OSCMessage` object, which is then sent out via UDP. Additionally, a simple client class exists that constructs the `OSCMessage` object for you.

2.1 Example

```
from pythonosc.udp_client import SimpleUDPClient

ip = "127.0.0.1"
port = 1337

client = SimpleUDPClient(ip, port)  # Create client

client.send_message("/some/address", 123)   # Send float message
client.send_message("/some/address", [1, 2., "hello"])  # Send message with int,␣
↪float and string
```

2.2 Client Module Documentation

CHAPTER 3

Server

The server receives OSC Messages from connected clients and invoked the appropriate callback functions with the dispatcher. There are several server types available.

3.1 Blocking Server

The blocking server type is the simplest of them all. Once it starts to serve, it blocks the program execution forever and remains idle inbetween handling requests. This type is good enough if your application is very simple and only has to react to OSC messages coming in and nothing else.

```
from pythonosc.dispatcher import Dispatcher
from pythonosc.osc_server import BlockingOSCUDPServer


def print_handler(address, *args):
    print(f"{address}: {args}")


def default_handler(address, *args):
    print(f"DEFAULT {address}: {args}")


dispatcher = Dispatcher()
dispatcher.map("/something/*", print_handler)
dispatcher.set_default_handler(default_handler)

ip = "127.0.0.1"
port = 1337

server = BlockingOSCUDPServer((ip, port), dispatcher)
server.serve_forever()  # Blocks forever
```

3.2 Threading Server

Each incoming packet will be handled in it's own thread. This also blocks further program execution, but allows concurrent handling of multiple incoming messages. Otherwise usage is identical to blocking type. Use for lightweight message handlers.

3.3 Forking Server

The process is forked every time a packet is coming in. Also blocks program execution forever. Use for heavyweight message handlers.

3.4 Async Server

This server type takes advantage of the asyncio functionality of python, and allows truly non-blocking parallel execution of both your main loop and the server loop. You can use it in two ways, exclusively and concurrently. In the concurrent mode other tasks (like a main loop) can run in parallel to the server, meaning that the server doesn't block further program execution. In exclusive mode the server task is the only task that is started.

3.4.1 Concurrent Mode

Use this mode if you have a main program loop that needs to run without being blocked by the server. The below example runs `init_main()` once, which creates the serve endpoint and adds it to the asyncio event loop. The transport object is returned, which is required later to clean up the endpoint and release the socket. Afterwards we start the main loop with `await loop()`. The example loop runs 10 times and sleeps for a second on every iteration. During the sleep the program execution is handed back to the event loop which gives the serve endpoint a chance to handle incoming OSC messages. Your loop needs to at least do an `await asyncio.sleep(0)` every iteration, otherwise your main loop will never release program control back to the event loop.

```
from pythonosc.osc_server import AsyncIOOSCUDPServer
from pythonosc.dispatcher import Dispatcher
import asyncio


def filter_handler(address, *args):
    print(f"{address}: {args}")


dispatcher = Dispatcher()
dispatcher.map("/filter", filter_handler)

ip = "127.0.0.1"
port = 1337


async def loop():
    """Example main loop that only runs for 10 iterations before finishing"""
    for i in range(10):
        print(f"Loop {i}")
        await asyncio.sleep(1)
```

(continues on next page)

(continued from previous page)

```
async def init_main():
    server = AsyncIOOSCUDPServer((ip, port), dispatcher, asyncio.get_event_loop())
    transport, protocol = await server.create_serve_endpoint()  # Create datagram
↪endpoint and start serving

    await loop()  # Enter main loop of program

    transport.close()  # Clean up serve endpoint


asyncio.run(init_main())
```

3.4.2 Exclusive Mode

This mode comes without a main loop. You only have the OSC server running in the event loop initially. You could of course use an OSC message to start a main loop from within a handler.

```
from pythonosc.osc_server import AsyncIOOSCUDPServer
from pythonosc.dispatcher import Dispatcher
import asyncio


def filter_handler(address, *args):
    print(f"{address}: {args}")


dispatcher = Dispatcher()
dispatcher.map("/filter", filter_handler)

ip = "127.0.0.1"
port = 1337

server = AsyncIOOSCUDPServer((ip, port), dispatcher, asyncio.get_event_loop())
server.serve()
```

3.5 Server Module Documentation

Python-osc implements a server and client for Open Sound Control. It allows a python access to a versatile protocol used in many musical instruments, controller hardware and control applications.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search