sc3nb

Thomas Hermann, Dennis Reinsch

Aug 23, 2023

OVERVIEW

CHAPTER

ONE

SC3NB

sc3nb is a python package that offers an interface to SuperCollider3 (SC3), with special support to be used within jupyter notebooks.

- Documentation
- Source code
- Bug reports

The goal of sc3nb is to facilitate the development of auditory displays and interactive sonifications by teaming up

- python (and particularly numpy, scipy, pandas, matplotlib etc.) for data science
- and SuperCollider3 for interactive real-time sound rendering.

It allows:

- to interface with the SuperCollider audio server (scsynth) aswell as the SuperCollider Language and Interpreter (sclang) via the SC class
- The SuperCollider audio server can be started and addressed via
  - OSC directly with OSC messages and bundles
  - Python implementations of Classes from SuperCollider like `Synth`, `SynthDef`, `Buffer` and `Bus`
  - the `Score` class for non-realtime synthesis
- use the SuperCollider language (sclang) interactively via a subprocess.
  - write SuperCollider language code in Jupyter Notebooks and let sclang evaluate it.
  - inject Python variables into your sclang code
  - get the results of the sclang code in Python
- helper functions such as linlin, cpsmidi, midicps, clip, ampdb, dbamp which work like their SC3 counterparts.

sc3nb can be used for

- multi-channel audio processing
- auditory display and sonification
- sound synthesis experiment
- audio applications in general such as games or GUI-enhancements
- signal analysis and plotting
- computer music and just-in-time music control
- any usecase that the SuperCollider 3 language supports

It is meant to grow into a backend for a sonification package, and can be used both from jupyter and in standard python software development.

1.1 Installation

- To use sc3nb you need a installation of SuperCollider on your system. See SuperCollider Download for installation files.
- To install sc3nb you can
  - install it locally in editable mode (i.e. changes to sc3nb code will automatically be "re-installed").
    - clone the repository from https://github.com/interactive-sonification/sc3nb
    - from inside the sc3nb directory run `pip install -e .`
  - or install it directly from PyPI using `pip install sc3nb`

1.2 Examples

We provide examples in the form of Jupyter notebooks. You see them executed in the User Guide section of the documentation and also download them from the sc3nb examples folder.

1.3 Publications & Citation

- A paper introducing sc3nb can be found at https://doi.org/10.1145/3478384.3478401
- The belonging supplementary material can be found at https://doi.org/10.4119/unibi/2956379
- The presentation of the paper can be found at https://www.youtube.com/watch?v=kuZZSNCS53E

If you use sc3nb please cite the sc3nb introduction paper https://doi.org/10.1145/3478384.3478401

CHAPTER

TWO

HOW DOES THIS WORK?

- **sclang** and **scsynth** are started as subprocesses and their outputs are collected.
- **scsynth** is then controlled via OSC
  - Direct control of the server shortcuts the detour via sclang and is both more efficient and promises a lower latency
- **sclang** communication is done with pipes (bi-directional) and with OSC for receiving return values

CHAPTER

THREE

SUPPORT, CAVEATS AND PROBLEMS

We strongly encourage you to share any problems that arise using sc3nb with us.

There are some things to consider when using sc3nb.

- The shortcut `ctrl/cmd + .` does currently only work in classic Jupyter notebooks, yet not in JupyterLab. It also will import sc3nb as `sc3nb` and thus you should avoid variables with the name `sc3nb`.
- We depend on the output to stdout of sclang and scsynth in some cases such as the startup of the server - or more obviously - for the sclang output when using `cmd()` with verbosity. This means if some language settings or updates of sclang/scsynth change the outputs to something that we don't expect, stuff will fail. However this should be quite easy to patch.
- There is an issue #104 in Jupyter that does leave sclang and scsynth running when restarting the Jupyter kernel. To avoid this we use `atexit` to cleanup the processes. However calling `exit()` on the SC instance should be preferred. To avoid conflicts with orphaned sclangs/scsynths we also look for leftover sclang/scsynth processes on starting either of them and we try to kill them. This might lead to killing sclang or scsynth processes that you wanted to keep alive. However, you can specify which parent processes are allowed for sclang/scsynth processes using the `allowed_parents` parameter.
- The SuperCollider objects are currently not guaranteed to be thread-safe, with the exception of Node (Synth & Group)
- Another thing about SuperCollider in general is, to keep in mind that there are limits in the network communication: currently we only support UDP and therefore have the corresponding limits. As an example, a Bundler will simply fail when you try to send too many messages.

CHAPTER

FOUR

USEFUL RESOURCES

For more information about SuperCollider, check out the following links:

- SuperCollider website
- SuperCollider on GitHub
- SuperCollider documentation and help
  - Client vs Server
  - Server Architecture - Main Design Concepts
  - Server Command Reference
  - Server Guide
  - Multi-client Setups
  - OSC Communication
- SuperCollider Language
  - sclang Startup File
  - Symbolic Notations
  - Operators
  - Syntax Shortcuts

CHAPTER

FIVE

CHANGELOG

5.1 Version 1.1.0

- Bundler Improvements
  - Improve Bundler.add and the respective example notebook
  - Add OSC Bundle splitting feature
- Reduce SC warning frequency
- Add curve argument to s1 SynthDef
- Use s1 instead of s2 in Server.blip
- Improve how the Sever init hooks are stored and allow removing them
- Save binary blobs of SynthDefs added using the SynthDef class
- Use pyamapping
- Update pre-commit hooks and GitHub actions
- Fix bugs and typos

See all changes

5.2 Version 1.0.2

- Improve Server.boot by checking first for already booted scsynth instance
- Improve ServerOptions hardware device specification
- Improve default SuperCollider installation paths for Windows
- Add Server.remote workaround for issue #15
- Fix a typo in node.py
- Improve docs
  - Add score notebook
  - Improve doc build and update CONTRIBUTING.md
- Update pre-commit hooks

See all changes

5.3 Version 1.0.1

- first PyPI release

The following section was generated from `examples/sc3nb-examples.ipynb`

CHAPTER

SIX

GETTING STARTED

6.1 Starting sc3nb

```
[ ]: import sc3nb as scn
```

To startup sc3nb (sclang, scsynth, and a python OSC server) use `startup` which will return a `SC` instance which is the central SuperCollider Interface

```
[ ]: sc = scn.startup()  # see Configuration of sc3nb startup below for more details
```

You can produce a test sound with `blip`, which should relax any tensions whether the server is up and running. This sound should be played by the default server start.

```
[ ]: sc.server.blip()
```

`sc` provides you the interfaces for

- `scsynth` via `sc.server`
- `sclang` via `sc.lang`

Configuration of sc3nb startup:

- Specifying the excutable location
- We try to find the sclang and scsynth executables in the `$PATH` environment variable and also in the default installation directories
  - On macOS they reside in `/Applications/SuperCollider.app/Contents` in the folders MacOS and Resources. To add these paths to your `$PATH`, simply add to your ~/.profile, e.g. (please adapt to your installation): `PATH=$PATH:/Applications/SuperCollider.app/Contents/MacOS:/Applications/SuperCollider.app/Contents/Resources`
  - On Windows they reside in your Installation folder f.e `C:\Program Files\SuperCollider-3.x.x`
- If the executables are not found, you can specify them with `sclang_path` / `scsynth_path` e.g. `sclang_path="/path/to/sclang-containing-dir/sclang"`
- You could also only use **scsynth** only and don't start **sclang** with `start_sclang=False`
- See `help(scn.startup)` for all options

```
[ ]: # help(scn.startup)
```

6.2 Basic examples

6.2.1 Use Python implementations to control the SuperCollider audio server

Create and control SuperCollider Synths

[ ]:
```
syn = scn.Synth("s2")
syn
```

[ ]:
```
syn.freq = 800
syn
```

[ ]:
```
syn.free()  # or use the ctrl-. (cmd-.) shortcut in Jupyter notebooks
syn
```

Send SynthDefs with python code injection

[ ]:
```
synth_dur = 0.5  # you can use python variables in SynthDefs by prepending ^
synth_def = scn.SynthDef('random', """{ |out|
    var line;
    line = Line.kr(1, 0, ^synth_dur, doneAction: Done.freeSelf);
    Out.ar(out, SinOsc.ar(Rand(400, 800), 0, 0.2) * line);
}""")
synth_def.add()
synth_def  # Note that the representation has synth_dur already injected
```

[ ]:
```
scn.Synth("random")
```

Load a file as Buffer and play it

[ ]:
```
buf = scn.Buffer().read("./media/blip.wav")
buf
```

[ ]:
```
buf.play()
```

[ ]:
```
buffer_data = buf.to_array()
print(buffer_data.shape)
buffer_data
```

More examples about the above used SuperCollider objects can be found in the respective User Guide sections * *Nodes (Synths and Groups)* * *SynthDef* * *Buffer*

6.2.2 Use OSC to control the SuperCollider audio server

Send OSC Messages with SuperCollider Commands and receive replies

[ ]:
```
sc.server.msg("/status")  # waits for reply message and returns it
```

We offer also high level versions for the commands.

[ ]:
```
sc.server.status()
```

Create OSC Bundles using SuperCollider Commands directly

[ ]:
```
with sc.server.bundler() as bundler:
    bundler.add(0.2, "/s_new", ["s1", -1, 0, 0])
    bundler.add(0.5, "/s_new", ["s1", -1, 0, 0, "freq", 800, "dur", 0.1])
bundler.messages()
```

or create OSC Bundles with the high level Python implementations

[ ]:
```
with sc.server.bundler() as bundler:
    bundler.wait(0.2)
    scn.Synth("s1")
    bundler.wait(0.3)
    scn.Synth("s1", {"freq": 800, "dur": 0.1})
bundler.messages()
```

More OSC communication examples

6.2.3 Use the SuperCollider Language from Python

Execute SuperCollider Language Code

[ ]:
```
breakfast = "eggs"
sc.lang.cmd('^breakfast.scramble;')
```

or via the **%sc** IPython magic

[ ]:
```
%sc ^breakfast.scramble
```

Get results from SuperCollider Language in python with **cmdg** or **%scg**

[ ]:
```
x = 5
value = %scg (1..^x)
print(f"received {value} with type {type(value)}")
```

More sclang in sc3nb examples

6.2.4 Stoping the synthesis

There are multiple options to stop the playing audio synthesis

[ ]:
```
%sc x = Synth.new("default")  // starting a Synth with sclang
```

- to stop all playing synths either use CMD-. (in Jupyter Command mode).
- It is a shortcut for the ´free_all´ method of the default server

[ ]:
```
sc.server.free_all()
```

- or you could also use sclang

[ ]:
```
%sc s.freeAll
```

6.2.5 Combined usage

It is also possible to mix the different interaction styles.

- create a Synth with sclang and get the nodeid of the Synth via the **%scg** IPython magic

[ ]:
```
nodeid = %scg x = Synth("default"); x.nodeID;
```

- create the corresponding Python Synth instance

[ ]:
```
synth = scn.Synth("default", nodeid=nodeid, new=False)
synth
```

- Inspect and control the Synth from Python

[ ]:
```
synth.synth_desc
```

[ ]:
```
synth.freq
```

[ ]:
```
synth.freq *= 2
synth.freq
```

[ ]:
```
synth
```

- send a OSC Message to free the Synth

[ ]:
```
sc.server.msg("/n_free", nodeid)
```

[ ]:
```
synth
```

6.2.6 Exiting sc3nb

To shut down the server and sclang subprocesses

```
[ ]: sc.exit()
```

6.3 Further information and advanced examples

For more details on the specific parts please refer to the corresponding sections of the User Guide.

- Usage of *SuperCollider Language (sclang) in sc3nb*
- Usage of the SuperCollider Objects python interface
- *Server*
- *Nodes (Synth and Group)*
- *SynthDef*
- *Buffer*
- *Bus*
- *Recorder*
- Usage of *common SuperCollider helper functions*
- Usage of *OSC in sc3nb*

```
[ ]:
```

examples/sc3nb-examples.ipynb ends here.

The following section was generated from examples/sclang-examples.ipynb

CHAPTER

SEVEN

USAGE OF SCLANG IN SC3NB

You can send commands and receive data directly from the SuperCollider Language

```
[ ]: import time
     import numpy as np

     import sc3nb as scn
```

```
[ ]: sc = scn.startup()
```

7.1 sclang command execution

To send sc3 commands (i.e. program strings) to the language, either use the following functions

- `cmd()` normal command sending.

```
[ ]: # sc.cmd(cmdstr, pyvars)
     sc.lang.cmd('"Hello World".postln')  # output will be printed
```

- `cmds()` silent version without (alias for `cmd(.., verbose=False)`)

```
[ ]: sc.lang.cmds('"Hello User".postln')  # check jupyter console for output
```

- `cmdg()` send command and get the output (alias for `cmd(.., get_return=True)`). More details *below*

```
[ ]: string = sc.lang.cmdg('"sc3nb".postln')
     print(f'We received the string = "{string}"')
```

or use the corresponding Magics in Jupyter

- Jupyter line magics `%sc, %scv, %scs, %scg, %scgv, %scgs`
- Jupyter cell magics `%%sc, %%scv, %%scs, %%scg, %%scgv, %%scgs`

which wrap the above functions: `cmd{v,g,s} = %sc{v,g,s}` and v=verbose, g=get, s=silent verbose is default, so `%sc=%scv`

Line magics can be placed within code just as the function calls as shown here:

```
[ ]: for p in range(1, 10):  # a bouncing ball
         %scs Synth.new(\s1, [\freq, 200]) // this is SC code so use // instead of #
         time.sleep(1/p)
```

Use raw python strings for multi-line sc3-programs:

[ ]:
```
sc.lang.cmd(r"""
Routine({
    x = 5.collect{ |i|
        0.2.wait;
        Synth.new(\default, [\freq, 50+(50*i)]);
    };
    1.wait;
    x.do{|e|
        e.release;
        0.1.wait;};
}).play;
""")
```

alternatively, you can use the cell magics

[ ]:
```
%%sc
Routine({
    x = 5.collect{ |i|
        0.2.wait;
        Synth.new(\default, [\freq, 50+(50*i)]);
    };
    1.wait;
    x.do{|e|
        e.release;
        0.1.wait;};
}).play;
```

Note that the code is executed in sclang and Python is returning directly after sending the command.

7.2 sclang command execution with python variable injection

Python variables can be injected into sc3 commands by using the ^ special:

The following examples demonstrates it by setting frequencies by using python variables

[ ]:
```
for p in range(1, 50):  # a tone ladder
    freq = 50 + p*3
    dur = np.log(p)
    position = np.sign(p-25)
    %scs Synth.new(\s1, [\freq, ^freq, \dur, ^dur, \pan, ^position])
    time.sleep(0.05)
```

This is injection is done with

[ ]:
```
help(scn.util.convert_to_sc)
```

Here are some conversion examples

[ ]:
```
python_list = [1,2,3,4]
%sc ^python_list.class
```

[ ]:
```
complex_py = 1+1j
%sc ^complex_py.class
```

[ ]:
```
symbol = r"\\python"
%sc ^symbol.class
```

When using the `cmd`, `cmdg` and `cmds` functions you can also provide a dictionary with variable names as keys and content as values (which can use other python vars or statements)

[ ]:
```
sc.lang.cmdv("^name1 / ^name2", pyvars={'name1': 9,'name2': 9*2})
```

Without providing pyvars, variables are searched in the users namespace.

[ ]:
```
freq = 5
rate = 6
sc.lang.cmdv("(^freq + 1) * (^rate + 1)")
```

alternatively via the magic this is done as:

[ ]:
```
%scv (^freq + 1) * (^rate + 1)
```

7.3 Getting sclang output in python

- To *get* the output of an sclang snippet into a python variable, use the cmdg function.
- The following example shows how to transfer a synth's nodeID

[ ]:
```
# start a Synth
sc.lang.cmd(r"""x = Synth.new(\default)""")
```

[ ]:
```
# get the nodeId to python
nodeID = sc.lang.cmdg("x.nodeID")
print(nodeID)
```

[ ]:
```
# use the nodeID to free the Synth via a message to scsynth audio server directly
sc.server.msg("/n_free", nodeID)
```

sc.cmdg(), resp. %scg return integers, floats, strings and lists * %scg can be assigned to a python variable within code

[ ]:
```
a = %scg 1234 + 23452
print(f"returned an {type(a)} of value {a}")
```

[ ]:
```
a = %scg 1234.5.squared
print(f"returned an {type(a)} of value {a}")
```

[ ]:
```
a = %scg "sonification".scramble
print(f"returned an {type(a)} of value {a}")
```

[ ]:
```
%scs ~retval = "sonification".scramble
%scg ~retval ++ "!"
```

You can combine your code in a single row.

[ ]:
```
scramble = %scg ~retval = "sonification".scramble; ~retval ++ "!";
scramble
```

[ ]:
```
a = %scg (1,1.1..2)
```

Note that floating-point numbers do only have limited precision

[ ]:
```
print(f"list with length: {len(a)}")
a
```

However they should be close

[ ]:
```
[round(num, 6) for num in a]
```

[ ]:
```
np.allclose(a, np.arange(1, 2, 0.1))
```

7.4 Some more usage examples

You can use the SuperCollider GUI features.

[ ]:
```
sc.lang.cmd(r"MouseX.help", timeout=10)
```

[ ]:
```
%sc {SinOsc.ar(MouseX.kr(200,400))}.play  // move mouse horizontally, CMD-. to stop
```

[ ]:
```
%sc s.scope()
```

[ ]:
```
sc.server.free_all()  # leaves the scope running
```

[ ]:
```
%%sc
{
    x = Synth.new(\s2, [\freq, 100, \num, 1]);
    250.do{|i|
        x.set(\freq, sin(0.2*i.pow(1.5))*100 + 200);
        0.02.wait;
    };
    x.free;
}.fork
```

[ ]:
```
sc.exit()
```

examples/sclang-examples.ipynb ends here.

CHAPTER

EIGHT

SUPERCOLLIDER OBJECTS

The following section was generated from `examples/supercollider-objects/server-examples.ipynb`

8.1 Server

```
[ ]: import sc3nb as scn
```

The `SCServer` class is the central interface for

- controlling the SuperCollider audio server process
- managing SuperCollider Objects
- using OSC for outgoing and incoming packets

To achieve all this the SCServer is

- registered as a client to the SuperCollider audio server process (scsynth) and exchanging SuperCollider Commands with the server process
- and also running an OSC server in python which can communicate via OSC with scsynth and sclang.

For information about how to communicate using OSC see the *OSC communication notebook*. This notebook focuses on using the `SCServer` class for interacting with scsynth

8.1.1 Starting the Server

The most convienent way is to use the default sc3nb `SCServer` instance

```
[ ]: sc = scn.startup()
```

```
[ ]: sc.server
```

However you can also use the SCServer class directly

The server connection can be created

- locally using `boot`, which will start a scsynth process and connect to it or
- remote using `remote` for connecting to an already running scsynth process

```
[ ]: serv = scn.SCServer()
serv
```

```
[ ]: serv.boot()
```

Notice how the `SCServer` always tries to boot using the default SuperCollider audio server port 57110. But this port is already used by `sc.server` and thus the `SCServer` tries to connect to the already running instance using `SCserver.remote`. This enables a user to share the same scsynth instance with other users and/or use it from other notebooks. If the port to be used is explicitly specified the `SCServer` instance will fail instead of connecting.

The `SCServer` will register to the scsynth process using `SCServer.notify()`

Let's look how many clients are allowed and what the `client_id`s and the corresponding `default_group`s of the SCServer instances are.

```
[ ]: print(f"The scsynth process of this SCServer instance allows {sc.server.max_logins}␣
     ↪clients to login.")
```

```
[ ]: print(f"sc.server has client id {sc.server.client_id} and the default Group {sc.server.
     ↪default_group}")
```

```
[ ]: print(f"serv has client id {serv.client_id} and the default Group {serv.default_group}")
```

However also note that the instances use different ports meaning they are able to independently send and receive OSC packets

```
[ ]: sc.server.connection_info()
```

and also note that `serv` is not connected to sclang but has the same connection info for scsynth.

```
[ ]: serv.connection_info()
```

A Synth running on the SuperCollider audio server will be visible to all connected clients

```
[ ]: serv_synth = scn.Synth("s2", {"amp": 0.05, "pan": -1, "freq": 100}, server=serv)
```

```
[ ]: default_synth = scn.Synth("s2", {"amp": 0.05, "pan": 1, "freq": 440})  # no need to␣
     ↪specify sc.server as server argument
```

This also includes sclang, which is another client of the scsynth process

```
[ ]: %sc ~sclang_synth = Synth("s2", [\amp, 0.1])
```

```
[ ]: sc.server.dump_tree()
```

This also means freeing all Synths at once can be done with each client

```
[ ]: sc.server.free_all()
     # serv.free_all()
     # %sc s.freeAll
```

```
[ ]: sc.server.dump_tree()
```

and quitting one server also quits the others.

```
[ ]: serv.quit()
```

[ ]:
```
sc.server
```

Let's reboot the default server

[ ]:
```
sc.server.reboot()
```

More information about multi client setups can be found in the SuperCollider documentation.

8.1.2 Configuring Server options

Startup options of the `SCServer` instance can be set via `ServerOptions`, which can be passed as argument when starting the `SCServer`

The default ServerOptions in sc3nb are:

[ ]:
```
scn.ServerOptions()
```

8.1.3 Getting Information

The `SCServer` instance provides various kinds of information

- What nodes are currently running

[ ]:
```
sc.server.dump_tree()
```

[ ]:
```
sc.server.query_tree()
```

- The current status of the server, acquired via the `/status` OSC command

[ ]:
```
sc.server.status()
```

which can also be accessed directly via properties

[ ]:
```
sc.server.nominal_sr
```

[ ]:
```
sc.server.num_synthdefs
```

- the version of the SC3 server process, acquired via the `/version` OSC command

[ ]:
```
sc.server.version()
```

- the address of the SuperCollider audio server

[ ]:
```
sc.server.addr
```

- The connection info

[ ]:
```
sc.server.connection_info()
```

- other runtime properties of the Server

[ ]:
```
sc.server.has_booted
```

```
[ ]: sc.server.is_running
```

```
[ ]: sc.server.client_id
```

```
[ ]: sc.server.max_logins
```

```
[ ]: sc.server.default_group
```

```
[ ]: sc.server.output_bus
```

```
[ ]: sc.server.input_bus
```

8.1.4 Controlling Volume

```
[ ]: syn = scn.Synth("s2")
```

```
[ ]: sc.server.volume
```

```
[ ]: scn.dbamp(sc.server.volume)
```

```
[ ]: sc.server.muted
```

```
[ ]: sc.server.muted = True
```

```
[ ]: sc.server.volume = -10.0
```

```
[ ]: scn.dbamp(sc.server.volume)
```

```
[ ]: sc.server.muted = False
```

```
[ ]: sc.server.volume = 0.0
```

```
[ ]: scn.dbamp(sc.server.volume)
```

```
[ ]: syn.free()
     syn.wait(timeout=1)
```

8.1.5 Server dumps

The Server process can dump information about

- incoming OSC packages. See console for output

```
[ ]: sc.server.dump_osc() # specify level=0 to deactivate
```

- currently running Nodes

[ ]:
```
sc.server.dump_tree()  # Notice how the OSC packet is now included in the output
```

[ ]:
```
sc.server.blip() # see dumped bundle for test sound on console
```

8.1.6 Make a test sound

The following methods produces the `SCServer` startup sound. The test sound should ease any anxiety whether the server is properly started/running

[ ]:
```
sc.server.blip()
```

8.1.7 Managing Nodes

- freeing all running nodes and reinitialize the server

[ ]:
```
sc.server.free_all()
```

[ ]:
```
sc.server.free_all(root=False)  # only frees the default group of this client
```

- send the `/clearSched` OSC command. This is automatically done when using `free_all`

[ ]:
```
sc.server.clear_schedule()
```

- Execute init hooks. This is also automatically done when using `free_all`, `init` or `connect_sclang`

[ ]:
```
sc.server.execute_init_hooks()
```

- Adding init hooks.

[ ]:
```
sc.server.send_default_groups
```

- Syncing the SuperCollider audio server by sending a `/sync` OSC command and waiting for the reply.

[ ]:
```
sc.server.sync()
```

8.1.8 Allocating IDs

The `SCServer` instance manages the IDs for Nodes, Buffers and Buses for the SuperCollider Objects via the following methods. These can also be used for getting suitable IDs when manually creating OSC packages.

- Get the IDs via the allocator.

[ ]:
```
ids = sc.server.buffer_ids.allocate(num=2)
ids
```

- Free the IDs after usage via the allocator.

[ ]:
```
sc.server.buffer_ids.free(ids)
```

There are allocators for

- Nodes - `sc.server.node_ids`
- Buffer - `sc.server.buffer_ids`
- Buses (Audio and Control) - `sc.server.audio_bus_ids`, `sc.server.control_bus_ids`

[ ]:
```
sc.server.reboot()
```

[ ]:
```
# example to see how consecutive buffer alloc works:
ids = sc.server.buffer_ids.allocate(num=5)
print("5 buffers:", ids)
sc.server.buffer_ids.free(ids[0:2])
print("freed buffers ", ids[0:2])
ids4 = sc.server.buffer_ids.allocate(num=4)
print("allocated 4 buffers:", ids4, "-> new numbers to be consecutive")
sc.server.sync()
ids2 = sc.server.buffer_ids.allocate(num=2)
print("allocated 2 buffers:", ids2, "-> using the two freed before")
```

8.1.9 Handling SynthDefs

The server offers the following methods for handling SynthDefs. These are shortcuts for the respective SynthDef methods.

```
sc.server.send_synthdef
sc.server.load_synthdef
sc.server.load_synthdefs
```

Refer to the *SynthDef guide* for more information about SynthDefs.

[ ]:
```
sc.exit()
```

examples/supercollider-objects/server-examples.ipynb ends here.

The following section was generated from `examples/supercollider-objects/node-examples.ipynb`

[ ]:
```
import time

import sc3nb as scn
from sc3nb import Synth
```

[ ]:
```
sc = scn.startup(with_blip=False)
```

8.2 Nodes (Synth and Group)

One of the most important objects in SuperCollider are Nodes.

To see all Nodes on the Server use

[ ]:
```
sc.server.dump_tree()
```

You can also get a Python object represenation of the current server state via.

[ ]:
```
root_node = sc.server.query_tree()
root_node
```

Note that this will send a `/g_queryTree` command and parse the `/g_queryTree.reply` response of the server. The resulting Group representation will currently not be updated unless you use `query_tree` again.

[ ]:
```
root_node.children[-1]
```

A node is either a *Synth* or a *Group* of nodes

8.2.1 Synth

8.2.1.1 Create and control a Synth

Create a new Synth

[ ]:
```
synth = Synth(name="s2", controls={"freq": 100})
synth
```

You can now hear the Synth playing and see it in the NodeTree

[ ]:
```
sc.server.default_group
```

[ ]:
```
sc.server.query_tree()
```

Free a synth

[ ]:
```
synth.free()
```

Start the synth again

[ ]:
```
synth.new()
```

[ ]:
```
synth.nodeid
```

Calling `new` on an already running synth will cause a SuperCollider Server error

[ ]:
```
synth.new()
time.sleep(0.5)  # wait some time to ensure the audio server has send the /fail OSC␣
↪message
```

Pause a synth

[ ]:
```
synth.run(False)
```

[ ]:
```
synth
```

Run a paused synth

[ ]:
```
synth.run() # default flag for run is True
```

You can also wait for a Synth

[ ]:
```
synth_with_duration = scn.Synth("s1", dict(dur=2))
# wait for the Synth to finish playing
synth_with_duration.wait()
```

8.2.1.2 Set / get Synth parameters

Set synth parameters, using any for the following `set` calls

```
set(key, value, ...)
set(list_of_keys_and_values])
set(dict)
```

[ ]:
```
synth.set("freq", 200)
```

[ ]:
```
synth.set(["freq", 30, "amp", 0.3])
```

[ ]:
```
synth.set({"freq": 130, "amp": 0.1})
```

The Synth does save its current control arguments in `current_controls`

[ ]:
```
synth.current_controls
```

However these are only cached values on the python object. Updating them will only affect the `new` call. See below why this can be useful.

[ ]:
```
synth.current_controls['freq'] = 440
synth
```

[ ]:
```
synth.free()
synth.new() # The new Synth will be created with freq = 440
```

To see what arguments can be set you can look at the `synth_desc`

[ ]:
```
synth.synth_desc
```

You can use `get` to see the current value. This will request the current value from the SuperCollider audio server

[ ]:
```
synth.get("freq")
```

[ ]:
```
synth.get("pan")
```

This will also update the cached values in `current_args`

[ ]:
```
synth
```

This is also possible directly with

[ ]:
```
synth.pan
```

Which can also be used for setting the argument

```
[ ]: synth.pan = -1
```

You can also query information about the Synth. Look at *Group* for more information about these values

```
[ ]: synth.query()
```

```
[ ]: synth.free()
```

Keep in mind that getting a synth value is always querying the server.

This means we need to receive a message which cant be done using the Bundler. If you want to use relative values in a Bundler you should use the cached values from `current_args`

```
[ ]: with sc.server.bundler() as bundle:
         synth.new({"freq": 600})
         for _ in range(100):
             synth.set(['freq', synth.current_controls['freq'] * 0.99])
             bundle.wait(0.05)
         synth.free()
```

```
[ ]: synth.wait() # wait for synth
```

Some methods also allow getting the OSC Message instead of sending

```
[ ]: synth.new(return_msg=True)
```

Refer to the *OSC communication example notebook* if you want to learn more about messages and bundles.

8.2.2 Group

Nodes can be grouped and controlled together. This gives you mutliple advantages like

- controlling multiple Synth together
- specifying the execution order of the nodes. For more details look at *Order of nodes*

The SuperCollider audio server scsynth does have one root Group with the Node ID 0.

In this root group each user got an default group were all the Nodes of the user (Synths and Groups) should be located

```
[ ]: sc.server.query_tree()
```

We have the following default Group

```
[ ]: sc.server.default_group
```

8.2.2.1 Creating Groups

```
[ ]: g0 = scn.Group()
     g0
```

```
[ ]: sc.server.dump_tree()
```

```
[ ]: g0.free()
```

```
[ ]: sc.server.dump_tree()
```

```
[ ]: g0.new()
```

```
[ ]: sc.server.query_tree()
```

```
[ ]: sc.server.default_group
```

Create a Group in our new Group

```
[ ]: g1 = scn.Group(target=g0)
     g1
```

```
[ ]: sc.server.query_tree()
```

You can get information about you Group via

```
[ ]: g0.query()
```

The query contains the same information as a Synth query and additionally the head and tail Node of this group.

Notice the special value `-1` meaning `None`.

```
[ ]: g1.query()
```

Free the node

```
[ ]: g0.free()
```

```
[ ]: sc.server.query_tree()
```

8.2.3 Order of Nodes

The execution of the Nodes does plays an important role. For more details look at the SuperCollider Documentation

The Node placement of a Nodes can be controlled by the instantiation arguments

- `add_action` - An AddAction that specifies where to put Node relative to the target

```
[ ]: list(scn.AddAction)
```

- `target` - The target of the AddAction
- `group` - The group were the Node will be placed

8.2.3.1 Example of Ordering Nodes

[ ]:
```
g0 = scn.Group()
g0
```

[ ]:
```
g1 = scn.Group(target=g0)
g1
```

[ ]:
```
s0 = scn.Synth(target=g0, controls={"freq": 200})
```

[ ]:
```
s1 = scn.Synth(add_action=scn.AddAction.BEFORE, target=s0, controls={"freq": 600})
```

[ ]:
```
s2 = scn.Synth(add_action=scn.AddAction.TO_TAIL, target=g1, controls={"freq": 1200})
```

[ ]:
```
sc.server.query_tree()
```

[ ]:
```
s1.move(scn.AddAction.BEFORE, s2)
```

[ ]:
```
sc.server.query_tree()
```

[ ]:
```
g0
```

[ ]:
```
g0.run(False)
```

[ ]:
```
g0.run(True)
```

[ ]:
```
g1.run(False)
g1.query_tree()
```

[ ]:
```
s0.freq, s1.freq, s2.freq
```

[ ]:
```
g0.set("freq", 100)
s0.freq, s1.freq, s2.freq
```

[ ]:
```
sc.server.dump_tree()
```

[ ]:
```
g1.run(True)
```

[ ]:
```
sc.server.query_tree()
```

[ ]:
```
g1.set("freq", 440)

s0.freq, s1.freq, s2.freq
```

[ ]:
```
g0.query_tree()
```

[ ]:
```
g0.move_node_to_tail(s1)
```

```
[ ]: g0.dump_tree()
```

```
[ ]: g1.set("freq", 800)

     s0.freq, s1.freq, s2.freq
```

```
[ ]: sc.server.default_group.query_tree()
```

```
[ ]: sc.server.free_all()
     sc.server.dump_tree()
```

```
[ ]: sc.exit()
```

```
[ ]:
```

examples/supercollider-objects/node-examples.ipynb ends here.

The following section was generated from examples/supercollider-objects/synthdef-examples.ipynb

```
[ ]: import time

     import sc3nb as scn
     from sc3nb import SynthDef, Synth
```

```
[ ]: sc = scn.startup()
```

8.3 SynthDef

SynthDef wraps and extends the flexibility of SuperCollider Synth Definitions

To see how to create and use a Synth from the SynthDef please also see the *Synth examples*

8.3.1 SynthDef creation

```
[ ]: synth_def = SynthDef('random',
     """{ |out|
         var osc, env, freq;
         freq = Rand(400, 800);
         osc = SinOsc.ar(freq, 0, 0.2);
         env = Line.kr(1, 0, 1, doneAction: Done.freeSelf);
         Out.ar(out, osc * env);
     }""")
```

Note that you can copy the representation to you SuperCollider IDE

```
[ ]: synth_def
```

SynthDefs are created via **sclang** when you add them.

[ ]:
```
synth_def.add()
```

[ ]:
```
Synth("random")
```

8.3.2 Loading SynthDefs

You can also load SynthDefs from `.scsynthdef` files or as bytes via the Server

The Server allows - to send binary SynthDef content with `send_synthdef`

[ ]:
```
sc.server.send_synthdef?
```

- load a SynthDef file with `load_synthdef`
- load a directory with SynthDef files with `load_directory`

[ ]:
```
sc.server.load_synthdef?
```

This can be used together with supriya for example.

Note: These synth are not known to **sclang**, therefore getting the SynthDesc won't work

[ ]:
```
try:
    from supriya.synthdefs import SynthDefBuilder, Envelope
    from supriya.ugens import SinOsc, EnvGen, Out
    from supriya import DoneAction
except:
    print("Example needs supriya pakage installed")
else:
    with SynthDefBuilder(name='supriya_synth', amplitude=0.3, frequency=440.0, gate=1.0)␣
↪as builder:
        source = SinOsc.ar(frequency=builder['frequency'])
        envelope = EnvGen.kr(done_action=DoneAction.FREE_SYNTH,
                             envelope=Envelope.asr(),
                             gate=builder['gate'])
        source = source * builder['amplitude']
        source = source * envelope
        out = Out.ar(bus=0, source=source)

    # Build SynthDef with supriya SynthDefBuilder
    supriya_synthdef = builder.build()
    # Compile SynthDef to binary SynthDef and send it to the server.
    sc.server.send_synthdef(supriya_synthdef.compile())

    with sc.server.bundler(0.1) as bundler:
        syn1 = scn.Synth(supriya_synthdef.actual_name)  # the SynthDef name is saved in␣
↪the supriya_synthdef
        bundler.wait(0.5)
        syn1.release(0.5)

        bundler.wait(1)

        syn2 = scn.Synth(supriya_synthdef.actual_name, controls={"frequency": 220})
```

(continues on next page)

(continued from previous page)

```
        bundler.wait(1)
        syn2.free()
```

8.3.3 SynthDef creation with context

the sc3nb SynthDef - different from standard sclang SynthDefs - allows to inject a number of context specifiers using mustache syntax, i.e. `{{my_context_specifier}}`. This would mark a place within the synth definition that can be replaced by some user-wished sclang code before adding the SynthDef to the server.

You can create a 'proto' synth definition with the SynthDef

[ ]:
```
synth_def_context = SynthDef(name="myKlank", definition=r"""
{ |out=0, amp=0.3, freq=440|
    var klank = DynKlank.ar(`[[1,2], [1,1], [1.4,1]], {{EXCITER}}, freq);
    Out.ar(out, amp*klank!^p_channels);
}""")
```

In this example a new definition named `"myKlank"` offers a dynamic context `{{EXCITER}}`" and 3 value contexts `freqs`, `rings`, and `channels`. Now we want to replace the dynamic context EXCITER with user-specific code.

- use `set_context()` to replace a context with specific code

[ ]:
```
synth_def_context.set_context("EXCITER", "Dust.ar(20)")
```

- the specific code can include other context specifier strings in turn
  - this basically allows to create chained processing in python-compiled synths defs.

Remarks: * The context mechanism is very general: * it can be used both for code and values * e.g. for an array specification or a UGen selection. * To set a value (e.g. number or array), consider to use the pyvars syntax, i.e. using the caret '^' variable value injection of python variables into sc code

The definition is complete up to the pyvars. So let's create a myKlank in SC!

[ ]:
```
p_channels = 2
```

[ ]:
```
synth_def_context  # will show ^p_channels value when defined
```

[ ]:
```
synth_name = synth_def_context.add(name="kdust")
```

[ ]:
```
# for testing, let's create a synth and stop after 1s
%scv x = Synth.new(^synth_name, [\freq, 200, \amp, 0.05])
time.sleep(1)
%scv x.free
```

Now let's create another synthdef with a WhiteNoise excitation, but as a 1-channel version

[ ]:
```
p_channels = 1
```

[ ]:
```
synth_def_context.reset()
knoise = synth_def_context.set_context("EXCITER", "WhiteNoise.ar(0.2)").add(name="knoise
↪")
knoise
```

[ ]:
```
# for testing, let's create a synth and stop after 1s
%scv x = Synth.new(^knoise, [\amp, 0.05, \freq, 100])
time.sleep(1)
%scv x.free
```

To delete an existing synthdef in sc you can use

```
synthDef.free(name)
```

(but don't do it now as the synthdef is used for examples below)

Remove all unused placeholders from current_def by

[ ]:
```
synth_def_context.reset()
print(f"SC code with context vars: \n {synth_def_context.current_def} \n")
synth_def_context.unset_remaining()
print(f"SC code with unset context vars: \n {synth_def_context.current_def} \n", )
```

Here you see, that the placeholder {{EXCITER}} has been deleted. * With this method you make sure, that you don't have any unused placeholders in the definition before creating a SynthDef. * Note, however, that your code might then not be functional…

8.3.3.1 Example creation of many SynthDefs

In some cases you want to create many SynthDefs with only a small change. You can use the SynthDefs object multiple time to do this. Here we want to create playbuf synthdefs for 1 to 10 channels: (Reuse of the synthdef object, which is defined above)

[ ]:
```
playBufsynthDef = SynthDef("playbuf_", """
    { |out=0, bufnum=1, rate=1, loop=0, pan=0, amp=0.3 |
            var sig = PlayBuf.ar(^num_channels, bufnum,
                rate*BufRateScale.kr(bufnum),
                loop: loop,
                doneAction: Done.freeSelf);
            Out.ar(out, Pan2.ar(sig, pan, amp))
    }"""
)
```

[ ]:
```
synthPlaybufs = {}
for num in [1,2,4,8]:
    synthPlaybufs[num] = playBufsynthDef.add(pyvars={"num_channels": num}, name=f
↪"playbuf_{num}")
```

Now you can access via `synthPlayBufs[2]` to the 2-ch playbuf etc.

[ ]:
```
synthPlaybufs[2]
```

8.3.3.2 Use-case: DynKlank Synths with controllable nr. of filters

A problem with synthdefs is that some parameters can only be set at compile time. E.g. * A DynKlank needs to know the maximum nr. of filters in its filter bank at SynthDef time. * A synth will need to know the channel count at SynthDef time

Contexts allow to define such synthDefs dynamically on demand.

The following code is a dynamic DynKlank whose data-controlled nr. of filters is determined via the SynthDef class. * nr of channels and nr. of filters in the filter bank are specified via pyvars * TODO: find a way how to set amps, rings, and harms on Synth.new

[ ]:
```
scn.SynthDef
```

[ ]:
```
synth_def = scn.SynthDef(name="myKlank", definition=r"""
{ |out=0, amp=0.3, freq=440|
    var klank, n, harms, amps, rings;
    harms = \harms.kr(Array.series(^p_nf, 1, 1));
    amps = \amps.kr(Array.fill(^p_nf, 0));
    rings = \rings.kr(Array.fill(^p_nf, 0.1));
    klank = DynKlank.ar(`[harms, amps, rings], {{EXCITER}}, freq);
    Out.ar(out, amp*klank!^p_channels);
}""")
```

[ ]:
```
# now create a synth where exciter is Dust, with 10 filters and stereo
kdust = synth_def.set_context("EXCITER", "Dust.ar(80)").add(
    pyvars={"p_nf": 10, "p_channels": 2})
print(kdust)
```

[ ]:
```
x = scn.Synth(name=kdust,
              controls={"freq": 100,
                        "amp": 0.05,
                        "harms": [60,60,60],
                        "amps": [0.1,0.1,0.1],
                        "rings": [1, 0.4, 0.2],})
```

[ ]:
```
x.set({"harms":[1,2,6], "amps": [0.1,0.1,0.1], "rings": [1, 0.4, 0.2]})
```

[ ]:
```
# following syntax works the same:
# x.set(["harms",[1,2,6],"amps",[0.1,0.1,0.1],"rings",[1, 0.4, 0.2]])
```

[ ]:
```
x.free()
```

8.3.4 Getting a SynthDesc

You can also query for a SynthDesc via **sclang** with the class method `get_desc("synth_name")`

Lets see for the SynthDefs included in sc3nb

- SynthDef "s1" which is a discrete sound event with parameters
  - frequency `freq`
  - duration `dur`
  - attack time `att`
  - amplitude `amp`
  - number of harmonics `num`
  - spatial panning `pan`

[ ]:
```
scn.SynthDef.get_description("s1")
```

- SynthDef "s2" which is a continuous synth with parameters
  - frequency `freq`
  - amplitude `amp`
  - number of harmonics `num`
  - spatial panning `pan`
  - Exponential lag `lg`
  - EnvGen gate `gate`, which allows the `Node.release` method

[ ]:
```
scn.SynthDef.get_description("s2")
```

[ ]:
```
sc.exit()
```

[ ]:
```

```

examples/supercollider-objects/synthdef-examples.ipynb ends here.

The following section was generated from `examples/supercollider-objects/buffer-examples.ipynb`

[ ]:
```
# header / imports
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import sc3nb as scn
```

[ ]:
```
from sc3nb import Buffer

example_file = "../media/blip.wav"
```

[ ]:
```
sc = scn.startup()
```

8.4 Buffer

Buffer is the Python class in sc3nb to interface with Buffers on the SuperCollider server.

[ ]:
```
# uncomment following line to see help for the Buffer class:
# help(scn.Buffer)
```

8.4.1 Create Buffer from a numpy.Array

[ ]:
```
d0 = np.random.rand(30000, 1)
```

[ ]:
```
buf0 = Buffer().load_data(d0)
buf0
```

In this case a default buffer with default sample rate (44100) and default insert mode is created.

If you want to create a buffer with a specific sample rate or OSC insertion method, etc. look at `load_data`

[ ]:
```
scn.Buffer.load_data?
```

Attention: insertion via OSC is particularly useful for small datasets (e.g. less than 1000 entries). For larger datasets the default 'file' mode is much faster.

[ ]:
```
d0 = np.random.rand(30000, 1)
buf1 = Buffer().load_data(d0, sr=5000, mode='osc')
buf1
```

8.4.2 Create Buffer with data from PyA Asig

This only works if using pya package: skip if you dont use pya

[ ]:
```
try:
    from pya import Ugen
except ImportError:
    pass
else:
    a1 = Ugen().sine(440, dur=1.0, sr=2000, channels=2).fade_out(0.5) # 1.0s sine tone␣
↪of 440 Hz
    a1.plot()
    print(a1)
    buf1 = Buffer().load_asig(a1)
    buf1
```

Again, default transport method is mode='file', i.e. using a temporary file and fill the buffer on sc with this content. * use mode="osc" to select the direct transfer of data via OSC messages

8.4.3 Create Buffer of .wav File

```
[ ]: buf2 = Buffer().read(example_file)
     buf2
```

The buffer method will automatically read the sample reate of the file and set it to Buffer.sr

You can specify further arguments to `read`

```
[ ]: scn.Buffer.read?
```

```
[ ]: buf = Buffer().read(example_file, starting_frame=18000, num_frames=20000, channels=[1])
     buf
```

8.4.4 Allocate an empty Buffer

```
[ ]: buf3 = Buffer().alloc(2.5*44100, sr=44100)
     buf3
```

8.4.5 Reuse an existing SC buffer

`Buffer.use_existing(bufnum)` will force the Buffer to (re-)use a buffer that already exists on the server, identified via its bufnum on the scsynth.

```
[ ]: # create a Buffer in SuperCollider
     %sc b = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01.wav");
```

```
[ ]: bufnum = %scg b.bufnum
     bufnum
```

```
[ ]: buf4 = Buffer()
     buf4
```

```
[ ]: buf4.use_existing(bufnum)
     buf4 # bufnum has now changed to be bufnum
```

```
[ ]: buf4.play()
```

8.4.6 Copy an existing SC buffer

`copy_existing` allows to copy an already existing buffer into another buffer.

```
[ ]: buf5 = Buffer().read(example_file)
     buf6 = Buffer().copy_existing(buf5)
```

This method will automatically use an intern SuperCollider copy method, if both buffer objects use the same sc instance. Otherwise the buffer will be loaded via filesystem. For this to happen, both sc instance should use the same filesystem.

```
[ ]: server2 = scn.SCServer(options=scn.ServerOptions(udp_port=57778))
     server2.boot(kill_others=False)
```

```
[ ]: sc.server.dump_osc()
```

```
[ ]: server2.dump_osc()
```

```
[ ]: buf7 = Buffer(server=server2).copy_existing(buf6)
```

```
[ ]: buf5sig = buf5.to_array()
     buf6sig = buf6.to_array()
     buf7sig = buf7.to_array()
     fig, axs = plt.subplots(4,1)
     axs[0].plot(buf5sig) # signal
     axs[1].plot(buf6sig) # copied signal
     axs[2].plot(buf7sig) # copied signal on other server
     axs[3].plot(buf6sig-buf7sig); # difference (should be 0)
     plt.tight_layout()
```

With this method, the complete buffer with all samples is copied. If you want to copy only a selection of samples, you can use `gen_copy()` (see below).

8.4.7 Play Buffer

If you want to listen to the buffer, you can use `play`.

```
[ ]: d = np.sin(2 * np.pi * 440 * np.linspace(0, 3, 3 * 44100)**0.9)
     buf8 = Buffer().load_data(d)
```

```
[ ]: playbuf_synth = buf8.play()
     playbuf_synth
```

As you can see `play()` returns an sc3nb Synth object for the Buffer.

This allows to control the playback via the synth class while the synth is running.

```
[ ]: playbuf_synth.rate = 0.5
```

```
[ ]: if not playbuf_synth.freed: # stop the playback if not done already
         playbuf_synth.free()
         playbuf_synth.wait()
```

```
[ ]: playbuf_synth = buf8.play(rate=10, amp=0.15, pan=1)  # play at given rate and pan
```

```
[ ]: playbuf_synth.wait(timeout=6)  # wait for synth to finish
```

You can get a description of the possible arguments with

```
[ ]: scn.SynthDef.get_description(playbuf_synth.name)
```

and even can see the SynthDef here:

[ ]:
```
buf8._synth_def
```

You can get a description of the possible arguments with

[ ]:
```
scn.SynthDef.get_description(playbuf_synth.name)
```

As you can see the SC synth will free itself when done if you are not using the loop argument.

However with loop enabled you need to free the synth manually.

[ ]:
```
synth = buf8.play(rate=-4, loop=True)  # play looped
```

[ ]:
```
synth.rate = 1 # change controls as needed
```

[ ]:
```
synth.free()
```

For more information regarding the Synth class, please refer to the *Node guide*.

8.4.8 Write Buffer content to file

Write the content of a buffer into a file. By default it is a .wav File with float as sample. You can change it via parameters "header" and "sample".

[ ]:
```
buf9 = Buffer().load_data(np.random.rand(10000)-0.5)
```

[ ]:
```
buf9.write("../media/output.wav")
```

[ ]:
```
# !ls -la ../media # uncomment if your shell offers ls
```

8.4.9 Fetch Buffer content to array

[ ]:
```
# create a buffer
buf2 = Buffer().read(example_file)
```

[ ]:
```
data = buf2.to_array()
```

[ ]:
```
plt.plot(data);
```

[ ]:
```
buf2.play(rate=1)
```

8.4.10 Fill Buffer with values

8.4.10.1 Fill a Buffer with zeros:

```
[ ]: scn.Buffer.zero?
```

```
[ ]: buf = Buffer().alloc(100)
     buf.zero()
     plt.plot(buf.to_array());
```

8.4.10.2 Fill a Buffer range with values:

```
[ ]: scn.Buffer.fill?
```

```
[ ]: buf = Buffer().alloc(500).fill(0, 90, 22).fill(200, 100, 5)
     plt.plot(buf.to_array());
```

Alternatively: fill buffer with single fill statement using multiple value triplets

```
[ ]: buf.fill([20, 50, -8000, 200, 100, 8000])
     plt.plot(buf.to_array());
```

8.4.10.3 Fill Buffer with sine wave harmonics of given amplitudes.

```
[ ]: scn.Buffer.gen_sine1?
```

```
[ ]: buf = Buffer().alloc(500).gen_sine1([1,-0.5,0,1.4,0,0,0.2])
     plt.plot(buf.to_array());
```

8.4.10.4 Fill Buffer with sine wave partials using specified frequencies and amplitudes.

```
[ ]: scn.Buffer.gen_sine2?
```

```
[ ]: buf = Buffer().alloc(1024).gen_sine2([[3.1, 1], [0.2, -2.5], [30, 0.3]])
     plt.plot(buf.to_array());
```

8.4.10.5 Fill Buffer with sinus waves and given frequency, amplitude, phase

```
[ ]: scn.Buffer.gen_sine3?
```

```
[ ]: buf = Buffer().alloc(1024).gen_sine3(
         [[1, 0.9, 1], [2, 0.3, +np.pi/2], [3, 0.3, 3]])
     plt.plot(buf.to_array());
```

8.4.10.6 Fill Buffer with series of chebyshev polynomials:

[ ]:
```
scn.Buffer.gen_cheby?
```

$\text{cheby}(n) = \text{amplitude} \cdot \cos(n \cdot \arccos(x))$

[ ]:
```
buf = Buffer().alloc(1024)
ch = [1]
for i in range(4):
    ch.insert(0, 0)
    buf.gen_cheby(ch)
    plt.plot(buf.to_array(), label=str(i));
plt.legend();
```

`gen_sine1` to `gen_sine3` and `gen_cheby` have the optional parameters: * **normalize**: Normalize peak amplitude of wave to 1.0. * **wavetable**: If set, then the buffer is written in wavetable format so that it can be read by interpolating oscillators. * **clear**: if set then the buffer is cleared before new partials are written into it. Otherwise the new partials are summed with the existing contents of the buffer.

8.4.10.7 Copy data of another Buffer:

[ ]:
```
scn.Buffer.gen_copy?
```

[ ]:
```
buf1 = Buffer().alloc(1024).fill(1024, 0, 0)
plt.plot(buf1.to_array());
buf2 = Buffer().alloc(1024).gen_sine1([1,0.5,0,1.4,0,0.5,0.2])

# copy samples 0..0+400 of buf2 into buf1 at position 2++
buf1.gen_copy(buf2, 0, 2, 400)
plt.plot(buf1.to_array());

# copy samples 250..end(=<0) of buf2 into buf1 at position 250++
buf1.gen_copy(buf2, 0, 250, 400)
plt.plot(buf1.to_array());
```

Here we copy 100 samples of `buf2` at starting pos 1 to buf3 at position 2. Use a negative amount of samples to copy all available samples

8.4.11 Get information about the Buffer

Information about the buffer object:

[ ]:
```
buf3
```

Information about the buffer in SC

[ ]:
```
buf3.query?
```

[ ]:
```
buf3.query()
```

8.4.12 Free Buffers

start with a buffer

```
[ ]: buf = Buffer().read(example_file)
     buf
```

```
[ ]: buf.query()  # works as intended
```

```
[ ]: buf.free()
```

```
[ ]: buf  # listed as not loaded, python Buffer instance still exists
```

```
[ ]: try:
         buf.query()  # raises an error after buf.free
     except RuntimeError:
         pass
     else:
         print("Buffer query on freed buffer should raise RuntimeError")
```

```
[ ]: sc.exit()
```

examples/supercollider-objects/buffer-examples.ipynb ends here.

The following section was generated from examples/supercollider-objects/bus-examples.ipynb

8.5 Bus

```
[ ]: import sc3nb as scn
```

```
[ ]: sc = scn.startup()
```

8.5.1 Using a Control Bus

```
[ ]: bus = scn.Bus('control')
     bus
```

```
[ ]: bus.get()
```

```
[ ]: syn = scn.Synth("s2")
```

map the frequency of the synth to the bus value

```
[ ]: syn.map("freq", bus)
```

```
[ ]: bus.set(120) # changes synths frequency
```

```
[ ]: bus.set(440)
```

Set the frequency value to remove the mapping

```
[ ]: syn.set("freq", 220)
```

```
[ ]: bus.set(440)  # no change
```

```
[ ]: syn.free()
```

Free the bus to mark the bus id as available again

```
[ ]: bus.free()
```

8.5.2 Use multiple Control Buses

```
[ ]: synth_name = scn.SynthDef("busMulti", """{ |out, freqs=#[200, 400], rates=#[1, 2] |
    Out.ar(out, Splay.ar(SinOsc.ar(freqs) * Decay2.ar(Impulse.ar(rates), 0.001, 0.1)) *␣
↪0.5);
}""").add()
```

```
[ ]: buses = scn.Bus("control", 2)
buses
```

```
[ ]: buses.get()
```

```
[ ]: syn = scn.Synth(synth_name)
```

```
[ ]: syn.map("rates", buses)
```

```
[ ]: buses.set(1, 1)
```

```
[ ]: buses.set(1, 2)
```

```
[ ]: buses.get()
```

```
[ ]: syn.free()
```

```
[ ]: buses.free()
```

```
[ ]: sc.server.free_all()
```

```
[ ]: sc.exit()
```

```
[ ]:
```

examples/supercollider-objects/bus-examples.ipynb ends here.

The following section was generated from `examples/supercollider-objects/recorder-examples.ipynb`

8.6 Recorder

[ ]:
```
import sc3nb as scn
```

[ ]:
```
sc = scn.startup()
```

[ ]:
```
help(scn.Recorder)
```

8.6.1 Recording sound into a file

[ ]:
```
# use the Recording class to capture the output
recorder = scn.Recorder(path="my_record.wav")

with sc.server.bundler() as bundler:
    recorder.start(0.1)
    # /s_new synth name, node id, add action (0 to head), target (1 default group),␣
↪synth arguments...
    scn.Synth("s1", {"freq": 200, "dur": 1})
    bundler.wait(0.3)
    scn.Synth("s1", {"freq": 300, "dur": 1})
    recorder.stop(1.5)
```

- note that the sorting in scsynth node tree is with 'at begin' rule
- otherwise the rendered tones would be rendered after the outbus was written to file resulting in an empty file.
- the file is located in the same folder as this .ipynb file.

[ ]:
```
sc.exit()
```

[ ]:
```

```

`examples/supercollider-objects/recorder-examples.ipynb` ends here.

The following section was generated from `examples/supercollider-objects/score-examples.ipynb`

8.7 Score

[ ]:
```
import sc3nb as scn
```

[ ]:
```
sc = scn.startup()
```

[ ]:
```
from sc3nb import Score, SynthDef
```

The Score class can be used for non-realtime synthesis.

- This is done by starting the SuperCollider audio server scsynth in the non-realtime mode.
- The server will read the provided OSC file and render the sound to the specified sound file.

- **Note** that this will require to send all required SynthDefs and Buffers at the beginning. However you can start using the Buffers & SynthDefs immediately after the corresponding OSCMessages as the audio server will handle all messages in the specified order.

The `Score.record_nrt` class method provides an easy interface that generates a OSC file from a dict with timings as keys and lists of `OSCMessages` as values.

[ ]:
```
help(Score.record_nrt)
```

Lets create a simple SynthDef for this demonstration

[ ]:
```
synthdef = SynthDef(
    "test",
    r"""{ |out, freq = 440|
            OffsetOut.ar(out,
                SinOsc.ar(freq, 0, 0.2) * Line.kr(1, 0, 0.5, doneAction: Done.freeSelf)
            )
        }""",
)
```

For creating the messages its recommended to use the Bundler class

[ ]:
```
with sc.server.bundler(send_on_exit=False) as bundler:
    synthdef.add()  # Send the test SynthDef
    bundler.add(0.0, "/s_new", ["test", 1003, 0, 0, "freq", 440])
    bundler.add(0.2, "/s_new", ["test", 1000, 0, 0, "freq", 440])
    bundler.add(0.4, "/s_new", ["test", 1001, 0, 0, "freq", 660])
    bundler.add(0.6, "/s_new", ["test", 1002, 0, 0, "freq", 220])
    bundler.add(1, "/c_set", [0, 0])  # The /c_set [0, 0] will close the audio file
```

The corresponding messages can be seen with

[ ]:
```
bundler.messages()
```

Lets start the non-realtime synthesis

[ ]:
```
Score.record_nrt(bundler.messages(), "../media/score.osc", "../media/score.wav", header_
↪format="WAV")
```

Lets listen to the created audio file with the IPython Audio class that allows to read and play audio files

[ ]:
```
from IPython.display import Audio
```

[ ]:
```
Audio("../media/score.wav")
```

[ ]:
```
sc.exit()
```

[ ]:
```

```

examples/supercollider-objects/score-examples.ipynb ends here.

The following section was generated from `examples/osc-communication-examples.ipynb`

[ ]:
```
import time
import numpy as np
```

[ ]:
```
import sc3nb as scn
```

CHAPTER

NINE

OSC COMMUNICATION

With the OSC communication module of sc3nb you can directly send and receive OSC packets.

Open Sound Control (OSC) is a networking protocol for sound and is used by SuperCollider to communicate between sclang and scsynth. sc3nb is itself a OSC client and server. This allows sc3nb to send and receive OSC traffic.

For more information on OSC and especially how Supercollider handles OSC packets please refer to the following links:

- Open Sound Control Specification
- Server vs Client SuperCollider Guide
- Server Command Reference
- SuperCollider Synth Server Architecture

[ ]:
```
sc = scn.startup()
```

sc3nb serves as OSC server and as client of the SuperCollider server `scsynth`. You can also communicate with the SuperCollider interpreter `sclang` via OSC.

You can see the current connection information with `sc.server.connection_info()`

[ ]:
```
(sc3nb_ip, sc3nb_port), receivers = sc.server.connection_info()
```

[ ]:
```
(sc3nb_ip, sc3nb_port), receivers
```

If you want to communicate via OSC with another receiver you could add its name via `sc.server.add_receiver(name: str, ip: str, port: int)`, or you can pass a custom receiver when sending OSC

[ ]:
```
sc.server.add_receiver("sc3nb", sc3nb_ip, sc3nb_port)
```

[ ]:
```
sc.server.connection_info()
```

9.1 Sending OSC

You can send OSC with

[ ]:
```
help(sc.server.send)
```

9.1.1 Messages

Use the `OSCMessage` or the `python-osc` package to build an OscMessage

[ ]:
```
scn.OSCMessage?
```

[ ]:
```
msg = scn.OSCMessage("/s_new", ["s1", -1, 1, 1,])
sc.server.send(msg)
```

A shortcut for sending Messages is

[ ]:
```
help(sc.server.msg)
```

[ ]:
```
sc.server.msg("/s_new", ["s1", -1, 1, 1,])
```

a more complex example

[ ]:
```
for p in [0,2,4,7,5,5,9,7,7,12,11,12,7,4,0,2,4,5,7,9,7,5,4,2,4,0,-1,0,2,-5,-1,2,5,4,2,4]:
    freq = scn.midicps(60+p)  # see helper fns below
    sc.server.msg("/s_new", ["s1", -1, 1, 0, "freq", freq, "dur", 0.5, "num", 1])
    time.sleep(0.15)
```

Note that the timing is here under python's control, which is not very precise. The `Bundler` class allows to do better.

Remarks:

- note that the python code returns immediately and all events remain in scsynth
- note that unfortunately scsynth has a limited buffer for OSC messages, so it is not viable to spawn thousends of events. scsynth will then simply reject OSC messages.
- this sc3-specific problem motivated (and has been solved with) TimedQueue, see below.

9.1.2 Bundles

[ ]:
```
from sc3nb.osc.osc_communication import Bundler
```

To send a single or multiple message(s) with a timetag as an OSC Bundle, you can use the `Bundler` class

- Bundlers allow to specify a timetag and thus let scsynth control the timing, which is much better, if applicable.
- A Bundler can be created as documented here

[ ]:
```
Bundler?
```

The prefered way of creating Bundlers for sending to the server is via

```
[ ]: help(sc.server.bundler)
```

This will add the `sc.server.latency` time to the timetag. By default this is `0.0` but you can set it.

```
[ ]: sc.server.latency
```

```
[ ]: sc.server.latency = 0.1
     sc.server.latency
```

A Bundler lets you add Messages and other Bundlers to the Bundler and accepts

- an OSCMessage or Bundler
- an timetag with an OSCMessage or Bundler
- or Bundler arguments like (timetag, msg_addr, msg_params) (timetag, msg_addr) (timetag, msg)

Also see *Nesting Bundlers* for more details on how Bundlers are nested

```
[ ]: help(Bundler.add)
```

add returns the Bundler for chaining

```
[ ]: msg1 = scn.OSCMessage("/s_new", ["s2", -1, 1, 1,])
     msg2 = scn.OSCMessage("/n_free", [-1])
     sc.server.bundler().add(1.5, msg1).add(1.9, msg2).send() # sound starts in 1.5s
```

An alternative is the usage of the *context manager*. This means you can use the `with` statement for better handling as follows:

```
[ ]: with sc.server.bundler() as bundler:
         bundler.add(0.0, msg1)
         bundler.add(0.3, msg2)
```

Instead of declaring the time explicitly with add you can also use `wait`.

```
[ ]: iterations = 3
     with sc.server.bundler() as bundler:
         for i in range(iterations):
             bundler.add(msg1)
             bundler.wait(0.3)
             bundler.add(msg2)
             bundler.wait(0.1)
```

This adds up the internal time passed of the Bundler

```
[ ]: bundler.passed_time
```

```
[ ]: assert bundler.passed_time == iterations * 0.3 + iterations * 0.1, "Internal time seems␣
     ↪wrong"
```

Here are some different styles of coding the same sound with the Bundler features

- server Bundler with add and Bundler Arguments

```
[ ]: with sc.server.bundler(send_on_exit=False) as bundler:
         bundler.add(0.0, "/s_new", ["s2", -1, 1, 1,])
         bundler.add(0.3, "/n_free", [-1])
```

```
[ ]: dg1 = bundler.to_raw_osc(0.0)  # we set the time_offset explicitly so all Bundle␣
     ↪datagrams are the same
     dg1
```

- Bundler with explict latency set and using add

```
[ ]: with Bundler(sc.server.latency, send_on_exit=False) as bundler:
         bundler.add(0.0, "/s_new", ["s2", -1, 1, 1])
         bundler.add(0.3, "/n_free", [-1])
```

```
[ ]: dg2 = bundler.to_raw_osc(0.0)
     dg2
```

- server Bundler with implicit latency and using automatic bundled messages (See *Automatic Bundling*)

```
[ ]: with sc.server.bundler(send_on_exit=False) as bundler:
         sc.server.msg("/s_new", ["s2", -1, 1, 1,], bundle=True)
         bundler.wait(0.3)
         sc.server.msg("/n_free", [-1], bundle=True)
```

```
[ ]: dg3 = bundler.to_raw_osc(0.0)
     dg3
```

```
[ ]: # assert that all created raw OSC datagrams are the same
     assert dg1 == dg2 and dg1 == dg3, "The datagrams are not the same"
```

Note: You can use the Bundler with the Synth and Group classes for easier Message creation. This also removes the burden of managing the IDs for the different commands.

Also make sure to look at the *Automatic Bundling Feature* which is using the bundled messages (`msg(..., bundle=True)`

```
[ ]: t0 = time.time()
     with sc.server.bundler() as bundler:
         for i, r in enumerate(np.random.randn(100)):
             onset = t0 + 3 + r
             freq = 500 + 5 * i
             bundler.add(onset, scn.Synth("s1",
                 {"freq": freq, "dur": 1.5, "num": abs(r)+1}, new=False
             ).new(return_msg=True))
```

9.1.2.1 Bundler Timestamp

Small numbers (<1e6) are interpreted as times in seconds relative to `time.time()`, evaluated at the time of sending

[ ]:
```
sc.server.bundler(0.5, "/s_new", ["s1", -1, 1, 0, "freq", 200, "dur", 1]).send()  # a␣
↪tone starts in 0.5s
sc.server.bundler(1.0, "/s_new", ["s1", -1, 1, 0, "freq", 300, "dur", 1]).send()  # a␣
↪tone starts in 1.0s
```

Attention:

Sending bundles with relative times could lead to unprecise timings. If you care about precision

- use a bundler with multiple messages (if you care about the timings relative to each other in one Bundler)
  - because all relative times of the inner messages are calculated on top of the outermost bundler timetag
- or provide an explict timetag (>1e6) to specify absolute times (see the following examples)

A single Bundler with multiple messages

[ ]:
```
bundler = sc.server.bundler()
bundler.add(0.5, "/s_new", ["s1", -1, 1, 0, "freq", 200, "dur", 1])
bundler.add(1.0, "/s_new", ["s1", -1, 1, 0, "freq", 300, "dur", 1])
bundler.send()  # second tone starts in 1.0s
```

using `time.time()+timeoffset` for absolute times

[ ]:
```
t0 = time.time()
sc.server.bundler(t0 + 0.5, "/s_new", ["s1", -1, 1, 0, "freq", 200, "dur", 1]).send()  #␣
↪a tone starts in 0.5s
sc.server.bundler(t0 + 1.0, "/s_new", ["s1", -1, 1, 0, "freq", 300, "dur", 1]).send()  #␣
↪a tone starts in 1.0s
```

[ ]:
```
t0 = time.time()
with sc.server.bundler() as bundler:
    for i, r in enumerate(np.random.randn(100)): # note: 1000 will give: msg too long
        onset = t0 + 3 + r
        freq = 500 + 5 * i
        msg_params = ["s1", -1, 1, 0, "freq", freq, "dur", 1.5, "num", abs(r)+1]
        bundler.add(onset, "/s_new", msg_params)
```

[ ]:
```
sc.server.free_all()
```

9.1.2.2 Nesting Bundlers

You can nest Bundlers: this will recalculate the time relative to the sending time of the outermost bundler

[ ]:
```
with sc.server.bundler() as bundler_outer:
    with sc.server.bundler(0.2) as bundler:
        sc.server.msg("/s_new", ["s2", -1, 1, 1,], bundle=True)
        bundler.wait(0.3)
        sc.server.msg("/n_free", [-1], bundle=True)
    bundler_outer.wait(0.8)
    bundler_outer.add(bundler)
```

```
[ ]: bundler_outer
```

Or you can nest using `Bundler.add` and get the same results

```
[ ]: bundler_outer_add = sc.server.bundler()
     bundler_outer_add.add(bundler)
     bundler_outer_add.add(0.8, bundler)
```

```
[ ]: assert bundler_outer.to_raw_osc(0.0) == bundler_outer_add.to_raw_osc(0.0), "Bundler␣
     ↪contents are not the same"
```

Notice that using relative timetags with wait will delay the relative timetags aswell.

```
[ ]: bundler_outer_add = sc.server.bundler()
     bundler_outer_add.wait(1)  # delays both bundles
     bundler_outer_add.add(bundler)
     bundler_outer_add.add(0.8, bundler)
```

```
[ ]: bundler_outer_add = sc.server.bundler()
     bundler_outer_add.add(bundler)
     bundler_outer_add.wait(1)  # delays the 2nd bundle
     bundler_outer_add.add(0.8, bundler)
```

This can be helpful in loops where the relative time then can be seen as relative for this iteration

```
[ ]: with sc.server.bundler() as nested_bundler_loop:
         for i in range(3):
             nested_bundler_loop.add(0.5, bundler)
             nested_bundler_loop.wait(1)
     nested_bundler_loop
```

9.1.3 Managing IDs

The OSC commands often require IDs for the different OSC commands. These should be manged by the `SCServer` to ensure not accidentally using wrong IDs.

- See Allocating IDs in the *Server guide* for more information about correctly using IDs manually.
- Or use *Automatic Bundling* and let sc3nb do the work for you!

9.1.4 Automatic Bundling

Probably the most convenient way of sending OSC is by using the Automatic Bundling Feature.

This allows you to simply use the SuperCollider Objects Synth and Group in the Context Manager of a Bundler and they will be automatically captured and stored.

```
[ ]: with sc.server.bundler() as bundler:
         synth = scn.Synth("s2")
         bundler.wait(0.3)
         synth.set("freq", 1000)
         bundler.wait(0.1)
```

(continues on next page)

(continued from previous page)

```
    synth.free()
synth.wait()
```

Note that it is important that to *only* wait on the Synth *after* the context of the Bundler has been closed.

If you'd call `synth.wait()` in the Bundler context, it would wait before sending the /s_new Message to the server and then wait forever (or until timeout) for the /n_end notification.

[ ]:
```
try:
    with sc.server.bundler() as bundler:
        synth = scn.Synth("s2")
        bundler.wait(0.3)
        synth.set("freq", 1000)
        bundler.wait(0.1)
        synth.free()
        synth.wait(timeout=2)  # without a timeout this would hang forever
except RuntimeError as error:
    print(error)
```

9.2 Receiving OSC packets

sc3nb is receiving OSC messages with the help of queues, one AddressQueue for each OSC address for which we want to receive messages.

[ ]:
```
sc.server.msg_queues
```

To see more information what messages are sent and received, set the logging level to INFO as demonstrated below.

[ ]:
```
import logging
logging.basicConfig(level=logging.INFO)
# even more verbose logging is avaible via
# logging.basicConfig(level=logging.DEBUG)
```

9.2.1 Getting replies

For certain outgoing OSC messages an incoming Message is defined.

This means that on sending such a message sc3nb automatically waits for the incoming message at the corresponding Queue and returns the result.

An example for this is `/sync {sync_id} -> /synced {sync_id}`

[ ]:
```
sc.server.msg("/sync", 12345)
```

See all `(outgoing message, incoming message)` pairs:

[ ]:
```
sc.server.reply_addresses
```

You can get the reply address via

[ ]:
```
sc.server.get_reply_address("/sync")
```

or

```
[ ]: sc.server.reply_addresses["/sync"]
```

If we specify `await_reply=False` the message will be kept in the queue

```
[ ]: sc.server.msg("/sync", 1, await_reply=False)
```

```
[ ]: sc.server.msg_queues[sc.server.get_reply_address("/sync")]
```

```
[ ]: sc.server.msg("/sync", 2, await_reply=False)
```

```
[ ]: sc.server.msg_queues[sc.server.reply_addresses["/sync"]]
```

```
[ ]: sc.server.msg_queues["/synced"]
```

You can see how many values were hold.

```
[ ]: sc.server.msg_queues["/synced"].skips
```

Notice that these hold messages will be skipped.

```
[ ]: sc.server.msg("/sync", 3, await_reply=True)
```

```
[ ]: sc.server.msg_queues["/synced"]
```

Therefore you should retrieve them with `get` and set `skip=False` if you care for old values in the queue and dont want them to be skipped.

```
[ ]: sc.server.msg("/sync", 42, await_reply=False)
     sc.server.msg_queues["/synced"].get(skip=False)
```

9.2.2 Custom Message Queues

If you want to get additional OSC Messages you need to create a custom MessageQueue

```
[ ]: from sc3nb.osc.osc_communication import MessageQueue
```

```
[ ]: help(MessageQueue)
```

```
[ ]: mq = MessageQueue("/test")
```

```
[ ]: sc.server.add_msg_queue(mq)
```

```
[ ]: sc.server.msg("/test", ["Hi!"], receiver="sc3nb")
```

```
[ ]: sc.server.msg_queues["/test"]
```

```
[ ]: sc.server.msg("/test", ["Hello!"], receiver="sc3nb")
```

```
[ ]: sc.server.msg_queues["/test"]
```

```
[ ]: sc.server.msg_queues["/test"].get()
```

```
[ ]: sc.server.msg_queues["/test"].get()
```

If you want to create a pair of an outgoing message that will receive a certain incomming message you need to specify it via the `out_addr` arugment of `add_msg_queue` or you could use the shortcut for this `add_msg_pairs`

```
[ ]: help(sc.server.add_msg_pairs)
```

```
[ ]: sc.server.add_msg_pairs({"/hi": "/hi.reply"})
```

Let's use `OSCdef` in sclang to send us replies.

```
[ ]: %%sc
OSCdef.newMatching("say_hi", {|msg, time, addr, recvPort| addr.sendMsg("/hi.reply",
↪"Hello there!")}, '/hi');
```

```
[ ]: sc.server.msg("/hi", receiver="sclang")
```

There is also the class MessageQueueCollection, which allows to create multiple MessageQueues for a multiple `address/subaddresses` combination

```
[ ]: from sc3nb.osc.osc_communication import MessageQueueCollection
```

```
[ ]: help(MessageQueueCollection)
```

```
[ ]: mqc = MessageQueueCollection("/collect", ["/address1", "/address2"])
sc.server.add_msg_queue_collection(mqc)
```

```
[ ]: mqc = MessageQueueCollection("/auto_collect")
sc.server.add_msg_queue_collection(mqc)
```

```
[ ]: sc.server.reply_addresses
```

```
[ ]: sc.server.msg_queues
```

```
[ ]: %%scv
OSCdef.newMatching("ab", {|msg, time, addr, recvPort| addr.sendMsg('/collect', '/address1
↪', "toast".scramble)}, '/address1');
```

```
[ ]: %%scv
OSCdef.newMatching("ab", {|msg, time, addr, recvPort| addr.sendMsg('/collect', '/address2
↪', "sonification".scramble)}, '/address2');
```

```
[ ]: sc.server.msg("/address1", receiver="sclang")
```

```
[ ]: sc.server.msg("/address2", receiver="sclang")
```

9.3 Examples

9.3.1 Creating an OSC responder and msg to sclang for synthesis

[ ]:
```
%%sc
OSCdef(\dinger, { | msg, time, addr, recvPort |
    var freq = msg[2];
    {Pulse.ar(freq, 0.04, 0.3)!2 * EnvGen.ar(Env.perc, doneAction:2)}.play()
}, '/ding')
```

[ ]:
```
with scn.Bundler(receiver=sc.lang.addr):
    for i in range(5):
        sc.server.msg("/ding", ["freq", 1000-5*i], bundle=True)
```

[ ]:
```
for i in range(5):
    sc.server.msg("/ding", ["freq", 1000-5*i], receiver=sc.lang.addr)
```

[ ]:
```
%scv OSCdef.freeAll()
```

[ ]:
```
sc.exit()
```

examples/osc-communication-examples.ipynb ends here.

The following section was generated from examples/helper-examples.ipynb

CHAPTER

TEN

HELPER FUNCTIONS

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

[ ]:
```
import sc3nb as scn
```

- SuperCollider coders are familiar and frequently use a number of useful converter functions
- the helper functions provide pythonic pendants namely currently for (to be extended):

10.1 linlin(x, x1, x2, y1, y2, clip)

- to linearly map x from between [x1, x2] to [y1, y2]
- no range check is done, clipping as specified (None, “min”, “max” or anything for “minmax”)

[ ]:
```
xs = np.linspace(1, 9, 100)
plt.figure(figsize=(15,2))
for i, clip in enumerate([None, "min", "max", "minmax"]):
    plt.subplot(1, 4, i+1);
    plt.plot(xs, scn.linlin(xs, 3, 7, 500, 300, clip))
```

10.2 midicps and cpsmidi

[ ]:
```
scn.midicps(69.2)  # convert MIDI note to cycles per second (cps) in [Hz]
```

[ ]:
```
scn.cpsmidi(440)   # and back to MIDI note (in float resolution)
```

10.3 clip(value, minimim, maximum)

[ ]:
```
xs = np.linspace(1,9,100)
plt.plot([scn.clip(x, 5, 7) for x in xs]);
```

10.4 ampdb and dbamp

[ ]:
```
# dbamp(db) converts dB value in amplitude, 0 dB = 1, '*2' \approx +6dB
dbs = np.linspace(-20, 20)
plt.plot(dbs, [scn.dbamp(d) for d in dbs]);
# plt.semilogy()
```

[ ]:
```
# ampdb(amp) converts an amplitude to dB, assuming 0dB=1
scn.ampdb(0.2)
```

[ ]:
```

```

examples/helper-examples.ipynb ends here.

The following section was generated from examples/timedqueue-examples.ipynb

CHAPTER

ELEVEN

TIMEDQUEUE

11.1 Motivation:

- for sonifications (or any sound composition) with precise timing, usually a large number of events need to be spawned at the exact time.
- doing this with bundles doesn't work as the OSC buffer of scsynth is limited
- it needs a TimedQueue where events can be added for time-precise dispatching
- In TimedQueue, a thread then simply checks what items are due and executes them
- allowing arbitrary functions as objects to be queued for execution enables both sound-specific usecases (e.g. sending OSC messages/bundles) and also other things such as visualization
- However, the functions should complete really quickly as otherwise the queue would run late and fail to process due events
- hence, it remains in the user's responsibility to be careful
- If, however, longer programs are needed, functions can be spawned as threads on execution

11.2 Basic Demo of TimedQueue

The following demo illustrate the core functionality with console print and sound. Please check the console / shell from which you have launched jupyter-notebook (i.e. your stdout)

[ ]:
```
import sys, os, time, random
import numpy as np

import sc3nb as scn
```

[ ]:
```
sc = scn.startup()
```

[ ]:
```
queue = scn.TimedQueue()
```

[ ]:
```
def myfun(x):
    os.write(1, "{}\n".format(x).encode())
    sys.stderr.flush()

def myblip(freq):
    sc.server.msg("/s_new", ["s1", -1, 1, 0, "freq", freq, "num", 3])
```

[ ]:
```
myfun(4)  # the number is written to stdout
```

[ ]:
```
myblip(700) # a tone should play
```

[ ]:
```
t0 = time.time()
for i in range(50):
    queue.put(t0+i*0.04, myblip, 800+1*7*i)
    queue.put(t0+i*0.04, myfun,  400+30*i)  # plots on stderr = console
print(time.time()-t0)
```

Note that the code returns immediately, allowing you to interact with jupyter. All executions are then done by means of the TimedQueue

[ ]:
```
queue.close()
```

11.3 TimedQueueSC

To allow easy and fast usage of SC messages and bundles `TimedQueueSC` was created

- `put_msg(onset, address, params)` allows to send a message from python at onset.
- `put_bundler(onset, bundler)` allows to send a bundler from python at onset.

11.4 TimedQueueSC example with synchronized sound and mpl plot

- This example shows how to highlight data points as they are played.
- However, the marking is reset for every new data point, i.e. data points are not highlighted as long as the corresponding sound lasts
- to achieve that, see code example below

Note that there are sometimes some strange effects with matplotlib event loop hickups in Mac, it is untested with Linux or Windows, any problem reports or suggested solutions are welcome.

[ ]:
```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib qt5
```

[ ]:
```
# create some test data
data = np.vstack((np.random.randn(50, 5), np.random.randn(100, 5)+3.5))
```

[ ]:
```
# create figure, axis, plots -> a window should open depicting two clusters
fig, ax = plt.subplots(1)  # create figure
mngr = plt.get_current_fig_manager(); mngr.window.setGeometry(1200, 0, 500, 400)
pldata, = ax.plot(data[:,1], data[:,2], ".", ms=5) # create plots
plmarked, = ax.plot([], [], "ro", ms=5, lw=0.5)
plt.show(block=False); plt.ion(); fig.canvas.draw() # not needed if plot shows

# create the queue
queue = scn.TimedQueueSC()
```

[ ]:
```
def update_plot(x, y):
    global fig, ax, pldata, plmarked
    plmarked.set_data([x], [y])
    ax.draw_artist(ax.patch)
    ax.draw_artist(pldata)
    ax.draw_artist(plmarked)
    fig.canvas.update() # additional fig.canvas.flush_events() not needed?
```

[ ]:
```
t0 = time.time()
for i, r in enumerate(data):
    onset = t0 + scn.linlin(r[1], data[:,1].min(), data[:,1].max(), 0.1, 4) + random.
↪random()*0.2 + 0.2
    freq = scn.midicps(scn.linlin(r[2], 2, 5, 60, 80))
    pos = scn.linlin(r[4], 0, 2, -1, 1)
    queue.put_bundler(onset-0.2, scn.Bundler(onset, "/s_new", ["s1", -1, 1, 0, "freq",␣
↪freq, "amp", 0.05, "dur", .52, "pos", pos]))
    queue.put(onset, update_plot, (r[1], r[2]), spawn=False)
print(f'time used: {time.time() - t0}')
```

Notice that any data point is turned red the moment its sound starts.

11.5 TimedQueueSC PMSon with matplotlib highlights

The following example illustrates how to use TimedQueues to maintain a 'currently playing selection' of data points, so that the GUI highlighting is deactivated when the corresponding sound stops

- this is achieved by scheduling a select and an unselect function at corresponding sound onset and stop time
- Note that here the plot update is done within a second loop of scheduled 'update_plot' invocations, at an frame rate independent of the sound events.

[ ]:
```
data = np.vstack((np.random.randn(300, 7), np.random.randn(300, 7)+5))
```

[ ]:
```
# create figure
fig, ax = plt.subplots(1)  # create figure
mngr = plt.get_current_fig_manager()
mngr.window.setGeometry(1200, 0, 500, 400)
plt.show()

# create the queue
queue = scn.TimedQueueSC()
```

[ ]:
```
def mapcol(row, stats, col, val_from, val_to):  # helper for mapping
    return scn.linlin(row[col], stats[col, 0], stats[col, 1], val_from, val_to)

def select(i):  #  highlight selection
    selected[i] = True

def unselect(i): # lowlight selection
    selected[i] = False
```

(continues on next page)

(continued from previous page)

```
def update_plot(xs, ys):
    global fig, ax, pldata, plmarked, selected
    plmarked.set_data(xs[selected], ys[selected])
    ax.draw_artist(ax.patch)
    ax.draw_artist(pldata)
    ax.draw_artist(plmarked)
    fig.canvas.flush_events()
    fig.canvas.update()

# parameter mapping sonification with GUI
tot_dur = 5  # total duration of the sonification
max_ev_dur = 5.5  # maximal event duration
delay = 1  # offset

stats = np.vstack((np.min(data, 0), np.max(data, 0))).T
selected = np.zeros(np.shape(data)[0], bool)

# create axis, plots
ax.clear()
plmarked, = ax.plot([], [], "ro", ms=4, lw=0.5)
pldata, = ax.plot(data[:,1], data[:,2], ".", ms=2) # create plots

t0 = time.time()

for i, r in enumerate(data):
    onset = t0 + delay + 5* i/800 # mapcol(r, stats, 3, 0, tot_dur)
    freq  = scn.midicps( mapcol(r, stats, 2, 60, 90))
    ev_dur = mapcol(r, stats, 4, 0.2, max_ev_dur)
    # sonification
    synth_args = ["s1", -1, 1, 0, "freq", freq, "amp", 0.05, "dur", ev_dur, "pos", pos]
    bundler = scn.Bundler(onset, "/s_new", synth_args)
    queue.put_bundler(onset-delay, bundler)
    # on/off events of marker highlight
    queue.put(onset, select, i)
    queue.put(onset + ev_dur, unselect, i)

# update plot at given rate from earliest to latext time
for t in np.arange(t0, t0+delay+tot_dur+ev_dur+1, 1/10):  # 1 / update rate
    queue.put(t, update_plot, (data[:,1], data[:,2]))
```

11.6 TimedQueueSC PMSon with timeseries data and matplotlib

The following example illustrates howto create a continuous sonification with concurrent plotting the time in a plot

- This presumes time-indexable data
- a 'maximum onset' variable is maintained to shutdown the continuously playing synths when done
- note that the highlight will only replot the marker, required time is thus independent of the amount of data plotted in the other plot.

[ ]:
```
ts = np.arange(0, 20, 0.01)
data = np.vstack((ts,
                  np.sin(2.5*ts) + 0.01*ts*np.random.randn(np.shape(ts)[0]),
                  0.08*ts[::-1]*np.cos(3.5*ts)**2)).T
```

[ ]:
```
# create figure
fig, ax = plt.subplots(1)  # create figure
mngr = plt.get_current_fig_manager(); mngr.window.setGeometry(1200, 0, 500, 400)

# create axis, plots
ax.clear()
plmarked, = ax.plot([], [], "r-", lw=1)
pldata1, = ax.plot(data[:,0], data[:,1], "-", ms=2) # create plot 1
pldata2, = ax.plot(data[:,0], data[:,2], "-", ms=2) # create plot 2
```

[ ]:
```
# create the queue
queue = scn.TimedQueueSC()

def mapcol(row, stats, col, val_from, val_to):  # helper for mapping
    return scn.linlin(row[col], stats[col, 0], stats[col, 1], val_from, val_to)

def update_plot(t):
    global fig, ax, pldata1, pldata2, plmarked, selected
    plmarked.set_data([t,t], [-10000, 10000])
    ax.draw_artist(ax.patch)
    ax.draw_artist(pldata1)
    ax.draw_artist(pldata2)
    ax.draw_artist(plmarked)
    fig.canvas.update()
    # fig.canvas.flush_events()

stats = np.vstack((np.min(data, 0), np.max(data, 0))).T
selected = np.zeros(np.shape(data)[0], bool)

# parameter mapping sonification with GUI
delay = 0.5
rate = 2

t0 = time.time()
queue.put_msg(t0, "/s_new", ["s2", 1200, 1, 0, "amp", 0])
queue.put_msg(t0, "/s_new", ["s2", 1201, 1, 0, "amp", 0])

max_onset = 0
latest_gui_onset = 0
gui_frame_rate = 60

ts = []
for i, r in enumerate(data[::2, :]):
    ts.append(time.time()-t0)
    if i==0: tmin = r[0]
    onset = (r[0]-tmin)/rate
    freq  = scn.midicps( mapcol(r, stats, 1, 60, 70))
```

(continues on next page)

(continued from previous page)

```
    freqR = 0.5 * scn.midicps( mapcol(r, stats, 2, 70, 80))

    # sonification
    tt = t0 + delay + onset
    if tt > max_onset: max_onset = tt
    bundler = scn.Bundler(tt)
    bundler.add(0, "/n_set", [1200, "freq", freq, "num", 4, "amp", 0.2, "pan", -1, "lg",␣
↪0])
    bundler.add(0, "/n_set", [1201, "freq", freqR, "num", 1, "amp", 0.1, "pan", 1])
    queue.put_bundler(tt-0.2, bundler)
    if tt > latest_gui_onset + 1/gui_frame_rate:  # not more than needed gui updates
        latest_gui_onset = tt
        queue.put(tt, update_plot, (r[0],), spawn=False)
queue.put_msg(max_onset, "/n_free", [1200])
queue.put_msg(max_onset, "/n_free", [1201])

# queue.join()
print(time.time()-t0)
```

[ ]:
```
sc.exit()
```

examples/timedqueue-examples.ipynb ends here.

CHAPTER

TWELVE

SC3NB

Package for interfacing SuperCollider.

Collection of Classes and functions for communicating with SuperCollider within python and jupyter notebooks, as well as playing recording and visualizing audio.

Examples

For example usage please refer to the user guide.

12.1 Package Contents

12.1.1 Function List

startup	Inits SuperCollider (scsynth, sclang) and registers ipython magics

12.1.2 Class List

SC	Create a SuperCollider Wrapper object.
SCServer	SuperCollider audio server representaion.
ServerOptions	Options for the SuperCollider audio server
SCLang	Class to control the SuperCollider Language Interpreter (sclang).
Node	Representation of a Node on SuperCollider.
Synth	Representation of a Synth on SuperCollider.
Group	Representation of a Group on SuperCollider.
AddAction	AddAction of SuperCollider nodes.
SynthDef	Wrapper for SuperCollider SynthDef
Buffer	A Buffer object represents a SuperCollider3 Buffer on scsynth
Bus	Represenation of Control or Audio Bus(es) on the SuperCollider Server
Score	
Recorder	Allows to record audio easily.
TimedQueue	Accumulates events as timestamps and functions.
TimedQueueSC	Timed queue with OSC communication.
OSCMessage	Class for creating messages to send over OSC
Bundler	Class for creating OSCBundles and bundling of messages

12.1.3 Content

sc3nb.**startup**(*start_server: bool = True*, *scsynth_path: Optional[str] = None*, *start_sclang: bool = True*, *sclang_path: Optional[str] = None*, *magic: bool = True*, *scsynth_options: Optional[sc3nb.sc_objects.server.ServerOptions] = None*, *with_blip: bool = True*, *console_logging: bool = False*, *allowed_parents: Sequence[str] = ALLOWED_PARENTS*, *timeout: float = 10*) → *SC*

Inits SuperCollider (scsynth, sclang) and registers ipython magics

Parameters

start_server
[bool, optional] If True boot scsynth, by default True

scsynth_path
[Optional[str], optional] Path of scscynth executable, by default None

start_sclang
[bool, optional] If True start sclang, by default True

sclang_path
[Optional[str], optional] Path of sclang executable, by default None

magic
[bool, optional] If True register magics to ipython, by default True

scsynth_options
[Optional[ServerOptions], optional] Options for the server, by default None

with_blip
[bool, optional] make a sound when booted, by default True

console_logging
[bool, optional] If True write scsynth/sclang output to console, by default False

allowed_parents
[Sequence[str], optional] Names of parents that are allowed for other instances of sclang/scsynth processes, by default ALLOWED_PARENTS

timeout
[float, optional] timeout in seconds for starting the executable, by default 10

Returns

SC
SuperCollider Interface class.

`class sc3nb.SC(*, start_server: bool = True, scsynth_path: Optional[str] = None, start_sclang: bool = True, sclang_path: Optional[str] = None, scsynth_options: Optional[sc3nb.sc_objects.server.ServerOptions] = None, with_blip: bool = True, console_logging: bool = True, allowed_parents: Sequence[str] = ALLOWED_PARENTS, timeout: float = 5)`

Create a SuperCollider Wrapper object.

Parameters

start_server
[bool, optional] If True boot scsynth, by default True.

scsynth_path
[Optional[str], optional] Path of scscynth executable, by default None.

start_sclang
[bool, optional] If True start sclang, by default True.

sclang_path
[Optional[str], optional] Path of sclang executable, by default None.

scsynth_options
[Optional[ServerOptions], optional] Options for the server, by default None.

with_blip
[bool, optional] Make a sound when booted, by default True.

console_logging
[bool, optional] If True write scsynth/sclang output to console, by default True.

allowed_parents
[Sequence[str], optional] Names of parents that are allowed for other instances of sclang/scsynth processes, by default ALLOWED_PARENTS.

timeout
[float, optional] timeout in seconds for starting the executables, by default 5

`default: Optional[SC]`

Default SC instance.

This will be used by all SuperCollider objects if no SC/server/lang is specified.

Overview:

get_default	Get the default SC instance
start_sclang	Start this SuperCollider language
start_server	Start this SuperCollider server
_try_to_connect	
__del__	
__repr__	Return repr(self).
exit	Closes SuperCollider and shuts down server

classmethod get_default() → *SC*

Get the default SC instance

Returns

SC
default SC instance

Raises

RuntimeError
If there is no default SC instance.

start_sclang(*sclang_path: Optional[str] = None*, *console_logging: bool = True*, *allowed_parents: Sequence[str] = ALLOWED_PARENTS*, *timeout: float = 5*)

Start this SuperCollider language

Parameters

sclang_path
[Optional[str], optional] Path of sclang executable, by default None

console_logging
[bool, optional] If True write scsynth/sclang output to console, by default True

allowed_parents
[Sequence[str], optional] Names of parents that are allowed for other instances of sclang/scsynth processes, by default ALLOWED_PARENTS

timeout
[float, optional] timeout in seconds for starting the executable, by default 5

start_server(*scsynth_options: Optional[sc3nb.sc_objects.server.ServerOptions] = None*, *scsynth_path: Optional[str] = None*, *console_logging: bool = True*, *with_blip: bool = True*, *allowed_parents: Sequence[str] = ALLOWED_PARENTS*, *timeout: float = 5*)

Start this SuperCollider server

Parameters

scsynth_options
[Optional[ServerOptions], optional] Options for the server, by default None

scsynth_path
[Optional[str], optional] Path of scscynth executable, by default None

console_logging
[bool, optional] If True write scsynth/sclang output to console, by default True

with_blip
[bool, optional] make a sound when booted, by default True

allowed_parents
[Sequence[str], optional] Names of parents that are allowed for other instances of sclang/scsynth processes, by default ALLOWED_PARENTS

timeout
[float, optional] timeout in seconds for starting the executable, by default 5

`_try_to_connect`()

`__del__`()

`__repr__`() → str
Return repr(self).

`exit`() → None
Closes SuperCollider and shuts down server

class sc3nb.**SCServer**(*options: Optional[ServerOptions] = None*)

Bases: *sc3nb.osc.osc_communication.OSCCommunication*

SuperCollider audio server representaion.

12.1.3.1 Parameters

options
[Optional[ServerOptions], optional] Options used to start the local server, by default None

Create an OSC communication server

Parameters

server_ip
[str] IP address to use for this server

server_port
[int] port to use for this server

default_receiver_ip
[str] IP address used for sending by default

default_receiver_port
[int] port used for sending by default

Overview:

boot	Start the Server process.
init	Initialize the server.
execute_init_hooks	Run all init hook functions.
connect_sclang	Connect sclang to the server
add_init_hook	Create and add a hook to be executed when the server is initialized
remove_init_hook	Remove a previously added init Hook
bundler	Generate a Bundler with added server latency.
blip	Make a blip sound
remote	Connect to remote Server
reboot	Reboot this server
ping	Ping the server.

continues on next page

Table 1 – continued from previous page

quit	Quits and tries to kill the server.
sync	Sync the server with the /sync command.
send_synthdef	Send a SynthDef as bytes.
load_synthdef	Load SynthDef file at path.
load_synthdefs	Load all SynthDefs from directory.
notify	Notify the server about this client.
free_all	Free all node ids.
clear_schedule	Send /clearSched to the server.
send_default_groups	Send the default groups for all clients.
mute	Mute audio
unmute	Set volume back to volume prior to muting
version	Server version information
status	Server status information
dump_osc	Enable dumping incoming OSC messages at the server process
dump_tree	Server process prints out current nodes
query_tree	Query all nodes at the server and return a NodeTree
_init_osc_communication	
_get_errors_for_address	
_log_repr	
_log_message	
_warn_fail	
__repr__	Return repr(self).

boot(*scsynth_path: Optional[str] = None*, *timeout: float = 5*, *console_logging: bool = True*, *with_blip: bool = True*, *kill_others: bool = True*, *allowed_parents: Sequence[str] = ALLOWED_PARENTS*)

Start the Server process.

Parameters

scsynth_path
[str, optional] Path of scscynth executable, by default None

timeout
[float, optional] Timeout for starting the executable, by default 5

console_logging
[bool, optional] If True write process output to console, by default True

with_blip
[bool, optional] make a sound when booted, by default True

kill_others
[bool] kill other SuperCollider server processes.

allowed_parents
[Sequence[str], optional] Names of parents that are allowed for other instances of scsynth processes that won't be killed, by default ALLOWED_PARENTS

Raises

ValueError
If UDP port specified in options is already used

ProcessTimeout
If the process fails to start.

`init`(*with_blip: bool = True*)

Initialize the server.

This adds allocators, loads SynthDefs, send default Groups etc.

Parameters

with_blip
[bool, optional] make a sound when initialized, by default True

`execute_init_hooks`() → None

Run all init hook functions.

This is automatically done when running free_all, init or connect_sclang.

Hooks can be added using add_init_hook

`connect_sclang`(*port: int*) → None

Connect sclang to the server

This will add the "sclang" receiver and execute the init hooks

Parameters

port
[int] Port of sclang (NetAddr.langPort)

`add_init_hook`(*fun: Callable*, **args: Any*, ***kwargs: Any*) → *Hook*

Create and add a hook to be executed when the server is initialized

Parameters

hook
[Callable[..., None]] Function to be executed

args
[Any, optional] Arguments given to function

kwargs
[Any, optional] Keyword arguments given to function

Returns

Hook
The created Hook

`remove_init_hook`(*hook:* Hook)

Remove a previously added init Hook

Parameters

hook
[Hook] the hook to be removed

`bundler`(*timetag=0*, *msg=None*, *msg_params=None*, *send_on_exit=True*)

Generate a Bundler with added server latency.

This allows the user to easly add messages/bundles and send it.

Parameters

timetag
[float] Time at which bundle content should be executed. This servers latency will be added upon this. If timetag <= 1e6 it is added to time.time().

msg_addr
[str] SuperCollider address.

msg_params
[list, optional]

List of parameters to add to message.
(Default value = None)

Returns

Bundler
bundler for OSC bundling.

`blip`() → None

Make a blip sound

`remote`(*address: str*, *port: int*, *with_blip: bool = True*) → None

Connect to remote Server

Parameters

address
[str] address of remote server

port
[int] port of remote server

with_blip
[bool, optional] make a sound when initialized, by default True

`reboot`() → None

Reboot this server

Raises

RuntimeError
If this server is remote and can't be restarted.

`abstract ping`()

Ping the server.

`quit`() → None

Quits and tries to kill the server.

`sync`(*timeout=5*) → bool

Sync the server with the /sync command.

Parameters

timeout
[int, optional]

Time in seconds that will be waited for sync.
(Default value = 5)

Returns

bool
True if sync worked.

`send_synthdef`(*synthdef_bytes: bytes*)

Send a SynthDef as bytes.

Parameters

synthdef_bytes
[bytes] SynthDef bytes

wait
[bool] If True wait for server reply.

`load_synthdef`(*synthdef_path: str*)

Load SynthDef file at path.

Parameters

synthdef_path
[str] Path with the SynthDefs

bundle
[bool] Wether the OSC Messages can be bundle or not. If True sc3nb will not wait for the server response, by default False

`load_synthdefs`(*synthdef_dir: Optional[str] = None*, *completion_msg: Optional[bytes] = None*) → None

Load all SynthDefs from directory.

Parameters

synthdef_dir
[str, optional] directory with SynthDefs, by default sc3nb default SynthDefs

completion_msg
[bytes, optional] Message to be executed by the server when loaded, by default None

`notify`(*receive_notifications: bool = True*, *client_id: Optional[int] = None*, *timeout: float = 1*) → None

Notify the server about this client.

This provides the client id and max logins info needed for default groups.

Parameters

receive_notifications
[bool, optional] Flag for receiving node notification from server, by default True

client_id
[int, optional] Propose a client id, by default None

timeout
[float, optional] Timeout for server reply, by default 1.0

Raises

RuntimeError
If server has too many users.

OSCCommunicationError
If OSC communication fails.

`free_all`(*root: bool = True*) → None

Free all node ids.

Parameters

root
[bool, optional] If False free only the default group of this client, by default True

`clear_schedule()`
Send /clearSched to the server.

This clears all scheduled bundles and removes all bundles from the scheduling queue.

`send_default_groups()` → None
Send the default groups for all clients.

`mute()` → None
Mute audio

`unmute()` → None
Set volume back to volume prior to muting

`version()` → *ServerVersion*
Server version information

`status()` → *ServerStatus*
Server status information

`dump_osc(level: int = 1)` → None
Enable dumping incoming OSC messages at the server process

Parameters

level
[int, optional] Verbosity code, by default 1 0 turn dumping OFF. 1 print the parsed contents of the message. 2 print the contents in hexadecimal. 3 print both the parsed and hexadecimal representations.

`dump_tree(controls: bool = True, return_tree=False)` → Optional[str]
Server process prints out current nodes

Parameters

controls
[bool, optional] If True include control values, by default True

return_tree
[bool, optional] If True return output as string, by default False

Returns

str
If return_tree this is the node tree string.

`query_tree(include_controls: bool = True)` → *sc3nb.sc_objects.node.Group*
Query all nodes at the server and return a NodeTree

Parameters

include_controls
[bool, optional] If True include control values, by default True

Returns

NodeTree
object containing all the nodes.

_init_osc_communication()

_get_errors_for_address(*address: str*)

_log_repr()

_log_message(*sender*, **params*)

_warn_fail(*sender*, **params*)

__repr__() → str

Return repr(self).

class sc3nb.**ServerOptions**(*udp_port: int = SCSYNTH_DEFAULT_PORT*, *max_logins: int = 6*, *num_input_buses: int = 2*, *num_output_buses: int = 2*, *num_audio_buses: int = 1024*, *num_control_buses: int = 4096*, *num_sample_buffers: int = 1024*, *publish_rendezvous: bool = False*, *block_size: Optional[int] = None*, *hardware_buffer_size: Optional[int] = None*, *hardware_sample_size: Optional[int] = None*, *hardware_input_device: Optional[str] = None*, *hardware_output_device: Optional[str] = None*, *other_options: Optional[Sequence[str]] = None*)

Options for the SuperCollider audio server

This allows the encapsulation and handling of the command line server options.

Overview:

__repr__	Return repr(self).

__repr__()

Return repr(self).

class sc3nb.**SCLang**

Class to control the SuperCollider Language Interpreter (sclang).

Creates a python representation of sclang.

Raises

NotImplementedError
When an unsupported OS was found.

Overview:

start	Start and initilize the sclang process.
init	Initialize sclang for sc3nb usage.
load_synthdefs	Load SynthDef files from path.
kill	Kill this sclang instance.
__del__	
__repr__	Return repr(self).
cmd	Send code to sclang to execute it.
cmdv	cmd with verbose=True
cmds	cmd with verbose=False, i.e. silent
cmdg	cmd with get_result=True
read	Reads SuperCollider output from the process output queue.
empty	Empties sc output queue.
get_synth_description	Get a SynthDesc like description via sclang's global SynthDescLib.
connect_to_server	Connect this sclang instance to the SuperCollider server.

start(*sclang_path: Optional[str] = None*, *console_logging: bool = True*, *allowed_parents: Sequence[str] = ALLOWED_PARENTS*, *timeout: float = 10*) → None

Start and initilize the sclang process.

This will also kill sclang processes that does not have allowed parents.

Parameters

sclang_path
[Optional[str], optional] Path with the sclang executable, by default None

console_logging
[bool, optional] If True log sclang output to console, by default True

allowed_parents
[Sequence[str], optional] parents name of processes to keep, by default ALLOWED_PARENTS

timeout
[float, optional] timeout in seconds for starting the executable, by default 10

Raises

SCLangError
When starting or initilizing sclang failed.

init()

Initialize sclang for sc3nb usage.

This will register the /return callback in sclang and load the SynthDefs from sc3nb.

This is done automatically by running start.

load_synthdefs(*synthdefs_path: Optional[str] = None*) → None

Load SynthDef files from path.

Parameters

synthdefs_path
[str, optional] Path where the SynthDef files are located. If no path provided, load default sc3nb SynthDefs.

kill() → int

Kill this sclang instance.

Returns

int
returncode of the process.

__del__()

__repr__() → str

Return repr(self).

cmd(*code: str*, *pyvars: Optional[dict] = None*, *verbose: bool = True*, *discard_output: bool = True*, *get_result: bool = False*, *print_error: bool = True*, *get_output: bool = False*, *timeout: int = 1*) → Any

Send code to sclang to execute it.

This also allows to get the result of the code or the corresponding output.

Parameters

code
[str] SuperCollider code to execute.

pyvars
[dict, optional] Dictionary of name and value pairs of python variables that can be injected via ^name, by default None

verbose
[bool, optional] If True print output, by default True

discard_output
[bool, optional] If True clear output buffer before passing command, by default True

get_result
[bool, optional] If True receive and return the evaluation result from sclang, by default False

print_error
[bool, optional] If this and get_result is True and code execution fails the output from sclang will be printed.

get_output
[bool, optional] If True return output. Does not override get_result If verbose this will be True, by default False

timeout
[int, optional] Timeout in seconds for code execution return result, by default 1

Returns

Any

if get_result=True,
Result from SuperCollider code, not all SC types supported. When type is not understood this will return the datagram from the OSC packet.

if get_output or verbose
Output from SuperCollider code.

if get_output and get_result=True
(result, output)

else
None

Raises

RuntimeError
If get_result is True but no OSCCommunication instance is set.

SCLangError
When an error with sclang occurs.

`cmdv`(*code: str*, ***kwargs*) → Any

cmd with verbose=True

`cmds`(*code: str*, ***kwargs*) → Any

cmd with verbose=False, i.e. silent

`cmdg`(*code: str*, ***kwargs*) → Any

cmd with get_result=True

`read`(*expect: Optional[str] = None*, *timeout: float = 1*, *print_error: bool = True*) → str

Reads SuperCollider output from the process output queue.

Parameters

expect
[Optional[str], optional] Try to read this expected string, by default None

timeout
[float, optional] How long we try to read the expected string in seconds, by default 1

print_error
[bool, optional] If True this will print a message when timed out, by default True

Returns

str
output from sclang process.

Raises

timeout
If expected output string could not be read before timeout.

`empty`() → None

Empties sc output queue.

`get_synth_description`(*synth_def*)

Get a SynthDesc like description via sclang's global SynthDescLib.

Parameters

synth_def
[str] SynthDef name

Returns

dict
{argument_name: SynthArgument(rate, default)}

Raises

ValueError
When SynthDesc of synth_def can not be found.

connect_to_server(*server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

Connect this sclang instance to the SuperCollider server.

This will set Server.default and s to the provided remote server.

Parameters

server
[SCServer, optional] SuperCollider server to connect. If None try to reconnect.

Raises

ValueError
If something different from an SCServer or None was provided

SCLangError
If sclang failed to register to the server.

class sc3nb.**Node**(*, *nodeid: Optional[int] = None*, *add_action: Optional[Union[AddAction, int]] = None*, *target: Optional[Union[Node, int]] = None*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

Bases: abc.ABC

Representation of a Node on SuperCollider.

Create a new Node

Parameters

nodeid
[int or None] This Nodes node id or None

add_action
[AddAction or corresponding int, optional] This Nodes AddAction when created in Server, by default None

target
[Node or int or None, optional] This Nodes AddActions target, by default None

server
[SCServer, optional] The Server for this Node, by default use the SC default server

Overview:

new	Create a new Node
_get_status_repr	
_set_node_attrs	Derive Node group from addaction and target
free	Free the node with /n_free.
run	Turn node on or off with /n_run.
set	Set a control value(s) of the node with n_set.
_update_control	
_update_controls	
fill	Fill ranges of control values with n_fill.
map	Map a node's control to read from a bus using /n_map or /n_mapa.
release	Set gate as specified.
query	Sends an n_query message to the server.
trace	Trace a node.
move	Move this node
register	Register to be watched.
unregister	Unregister to stop being watched.
on_free	Callback that is executed when this Synth is freed
wait	Wait until this Node is freed
_parse_info	
_handle_notification	
__eq__	Return self==value.
_get_nodeid	Get the corresponding node id

abstract new(**args*, *add_action: Optional[Union[*AddAction*, int]] = None*, *target: Optional[Union[*Node*, int]] = None*, *return_msg: bool = False*, ***kwargs*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Create a new Node

Parameters

add_action
[AddAction or int, optional] Where the Node should be added, by default AddAction.TO_HEAD (0)

target
[Node or int, optional] AddAction target, if None it will be the default group of the server

_get_status_repr() → str

_set_node_attrs(*target: Optional[Union[*Node*, int]] = None*, *add_action: Optional[Union[*AddAction*, int]] = None*) → None

Derive Node group from addaction and target

Parameters

target
[int or Node] Target nodeid or Target Node of this Node's AddAction

add_action
[AddAction] AddAction of this Node, default AddAction.TO_HEAD (0)

free(*return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Free the node with /n_free.

This will set is_running and is_playing to false. Even when the message is returned to mimic the behavior of the SuperCollider Node See https://doc.sccode.org/Classes/Node.html#-freeMsg

Returns

Node or OSCMessage
self for chaining or OSCMessage when return_msg=True

run(*on: bool = True*, *return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Turn node on or off with /n_run.

Parameters

on
[bool] True for on, False for off, by default True

Returns

Node or OSCMessage
self for chaining or OSCMessage when return_msg=True

set(*argument: Union[str, Dict, List]*, **values: Any*, *return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Set a control value(s) of the node with n_set.

Parameters

argument
[str | dict | list] if string: name of control argument if dict: dict with argument, value pairs if list: use list as message content

value
[any, optional] only used if argument is string, by default None

Examples

```
>>> synth.set("freq", 400)
>>> synth.set({"dur": 1, "freq": 400})
>>> synth.set(["dur", 1, "freq", 400])
```

_update_control(*control: str*, *value: Any*) → None

_update_controls(*controls: Optional[Dict[str, Any]] = None*) → None

fill(*control: Union[str, int]*, *num_controls: int*, *value: Any*, *return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Fill ranges of control values with n_fill.

Parameters

control
[int or string] control index or name

num_controls
[int] number of control values to fill

value
[float or int] value to set

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

OSCMessage
if return_msg else self

map(*control: Union[str, int]*, *bus:* sc3nb.sc_objects.bus.Bus, *return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Map a node's control to read from a bus using /n_map or /n_mapa.

Parameters

control
[int or string] control index or name

bus
[Bus] control/audio bus

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

OSCMessage
if return_msg else self

release(*release_time: Optional[float] = None*, *return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Set gate as specified.

https://doc.sccode.org/Classes/Node.html#-release

Parameters

release_time
[float, optional] amount of time in seconds during which the node will release. If set to a value <= 0, the synth will release immediately. If None using its Envs normal release stage(s)

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

OSCMessage
if return_msg else self

query() → Union[*SynthInfo*, *GroupInfo*]

Sends an n_query message to the server.

The answer is send to all clients who have registered via the /notify command. Content of answer:

node ID the node's parent group ID previous node ID, -1 if no previous node. next node ID, -1 if no next node. 1 if the node is a group, 0 if it is a synth

if the node is a group:
ID of the head node, -1 if there is no head node. ID of the tail node, -1 if there is no tail node.

Returns

SynthInfo or GroupInfo
n_info answer. See above for content description

`trace`(*return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Trace a node.

Print out values of the inputs and outputs for one control period. If node is a group then print the node IDs and names of each node.

Parameters

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

Node or OSCMessage
if return_msg else self

`move`(*add_action: AddAction*, *another_node: Node*, *return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Move this node

Parameters

add_action
[AddAction [TO_HEAD, TO_TAIL, AFTER, BEFORE]] What add action should be done.

another_node
[Node] The node which is the target of the add action

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

Node or OSCMessage
if return_msg this will be the OSCMessage, else self

Raises

ValueError
If a wrong AddAction was provided

`abstract register`()

Register to be watched.

`abstract unregister`()

Unregister to stop being watched.

`on_free`(*func*)

Callback that is executed when this Synth is freed

`wait`(*timeout: Optional[float] = None*) → None

Wait until this Node is freed

Raises

TimeoutError
If timeout was provided and wait timed out.

_parse_info(*nodeid: int*, *group: int*, *prev_nodeid: int*, *next_nodeid: int*, **rest: Sequence[int]*) → Union[*SynthInfo*, *GroupInfo*]

_handle_notification(*kind: str*, *info*) → None

__eq__(*other*)

Return self==value.

static _get_nodeid(*value: Union[*Node*, int]*) → int

Get the corresponding node id

Parameters

value
[Node or int] If a Node is provided it will get its nodeid If a int is provided it will be returned

Returns

int
nodeid

Raises

ValueError
When neither Node or int was provided

class sc3nb.Synth(*name: Optional[str] = None*, *controls: Dict[str, Any] = None*, ***, *nodeid: Optional[int] = None*, *new: bool = True*, *add_action: Optional[Union[*AddAction*, int]] = None*, *target: Optional[Union[*Node*, int]] = None*, *server: Optional[*sc3nb.sc_objects.server.SCServer*] = None*)

Bases: *Node*

Representation of a Synth on SuperCollider.

Create a Python representation of a SuperCollider synth.

Parameters

name
[str, optional] name of the synth to be created, by default "default"

controls
[dict, optional] synth control arguments, by default None

nodeid
[int, optional] ID of the node in SuperCollider, by default sc3nb will create one. Can be set to an existing id to create a Python instance of a running Node.

new
[bool, optional] True if synth should be created on the server, by default True Should be False if creating an instance of a running Node.

add_action
[AddAction or int, optional] Where the Synth should be added, by default AddAction.TO_HEAD (0)

target
[Node or int, optional] AddAction target, if None it will be the default group of the server

server
[SCServer] sc3nb SCServer instance

Raises

ValueError
Raised when synth can't be found via SynthDescLib.global

Examples

```
>>> scn.Synth(sc, "s1", {"dur": 1, "freq": 400})
```

Overview:

_update_synth_state	
new	Creates the synth on the server with s_new.
get	Get a Synth argument
seti	Set part of an arrayed control.
__getattr__	
__setattr__	Implement setattr(self, name, value).
__repr__	Return repr(self).

_update_synth_state(*name: Optional[str]*, *controls: Optional[dict]*)

new(*controls: Optional[dict] = None*, *add_action: Optional[Union[*AddAction*, int]] = None*, *target: Optional[Union[*Node*, int]] = None*, *, *return_msg: bool = False*) → Union[*Synth*, *sc3nb.osc.osc_communication.OSCMessage*]

Creates the synth on the server with s_new.

Attention: Here you create an identical synth! Same nodeID etc. - This will fail if there is already this nodeID on the SuperCollider server!

get(*control: str*) → Any

Get a Synth argument

This will request the value from scsynth with /s_get(n).

Parameters

control
[str] name of the Synth control argument

abstract seti(**args*)

Set part of an arrayed control.

__getattr__(*name*)

__setattr__(*name*, *value*)

Implement setattr(self, name, value).

__repr__() → str

Return repr(self).

class sc3nb.**Group**(*, *nodeid: Optional[int] = None*, *new: bool = True*, *parallel: bool = False*, *add_action:* AddAction *= AddAction.TO_HEAD*, *target: Optional[Union[*Node*, int]] = None*, *server: Optional[*sc3nb.sc_objects.server.SCServer*] = None*)

Bases: *Node*

Representation of a Group on SuperCollider.

Create a Python representation of a SuperCollider group.

Parameters

nodeid
[int, optional] ID of the node in SuperCollider, by default sc3nb will create one. Can be set to an existing id to create a Python instance of a running Node.

new
[bool, optional] True if synth should be created on the server, by default True Should be False if creating an instance of a running Node.

parallel
[bool, optional] If True create a parallel group, by default False

add_action
[AddAction or int, optional] Where the Group should be added, by default AddAction.TO_HEAD (0)

target
[Node or int, optional] AddAction target, if None it will be the default group of the server

server
[SCServer, optional] Server instance where this Group is located, by default use the SC default server

Overview:

_update_group_state	
new	Creates the synth on the server with g_new / p_new.
move_node_to_head	Move node to this groups head with g_head.
move_node_to_tail	Move node to this groups tail with g_tail.
free_all	Frees all nodes in the group with g_freeAll.
deep_free	Free all synths in this group and its sub-groups with g_deepFree.
dump_tree	Posts a representation of this group's node subtree with g_dumpTree.
query_tree	Send a g_queryTree message for this group.
*_repr_pretty_*	
__repr__	Return repr(self).

_update_group_state(*children: Optional[Sequence[*Node*]] = None*) → None

new(*add_action=AddAction.TO_HEAD*, *target=None*, ***, *parallel=None*, *return_msg=False*) → Union[*Group*, *sc3nb.osc.osc_communication.OSCMessage*]

Creates the synth on the server with g_new / p_new.

Attention: Here you create an identical group! Same nodeID etc. - This will fail if there is already this nodeID on the SuperCollider server!

Parameters

add_action
[AddAction or int, optional] where the group should be added, by default AddAction.TO_HEAD (0)

target
[Node or int, optional] add action target, by default 1

parallel
[bool, optional] If True use p_new, by default False

return_msg
[bool, optional] If ture return the OSCMessage instead of sending it, by default False

Returns

Group
self

`move_node_to_head`(*node*, *return_msg=False*)

Move node to this groups head with g_head.

Parameters

node
[Node] node to move

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

Group
self

`move_node_to_tail`(*node*, *return_msg=False*)

Move node to this groups tail with g_tail.

Parameters

node
[Node] node to move

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

Group
self

`free_all`(*return_msg=False*)

Frees all nodes in the group with g_freeAll.

Parameters

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

OSCMessage
if return_msg else self

`deep_free`(*return_msg=False*)

Free all synths in this group and its sub-groups with g_deepFree.

Sub-groups are not freed.

Parameters

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

OSCMessage
if return_msg else self

`dump_tree`(*post_controls=True*, *return_msg=False*)

Posts a representation of this group's node subtree with g_dumpTree.

Parameters

post_controls
[bool, optional] True for control values, by default False

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

OSCMessage
if return_msg else self

`query_tree`(*include_controls=False*) → *Group*

Send a g_queryTree message for this group.

See https://doc.sccode.org/Reference/Server-Command-Reference.html#/g_queryTree for details.

Parameters

include_controls
[bool, optional] True for control values, by default False

Returns

tuple
/g_queryTree.reply

`_repr_pretty_`(*printer*, *cylce*)

`__repr__`() → str

Return repr(self).

`class sc3nb.AddAction`

Bases: enum.Enum

AddAction of SuperCollider nodes.

This Enum contains the codes for the different ways to add a node.

`TO_HEAD = 0`

`TO_TAIL = 1`

`BEFORE = 2`

`AFTER = 3`

`REPLACE = 4`

class sc3nb.**SynthDef**(*name: str*, *definition: str*, *sc: Optional[sc3nb.sc.SC] = None*)

Wrapper for SuperCollider SynthDef

Create a dynamic synth definition in sc.

Parameters

name
[string] default name of the synthdef creation. The naming convention will be name+int, where int is the amount of already created synths of this definition

definition
[string] Pass the default synthdef definition here. Flexible content should be in double brackets ("...{{flexibleContent}}..."). This flexible content, you can dynamic replace with set_context()

sc
[SC object] SC instance where the synthdef should be created, by default use the default SC instance

synth_descs

synth_defs

Overview:

get_description	Get Synth description
send	Send a SynthDef as bytes.
load	Load SynthDef file at path.
load_dir	Load all SynthDefs from directory.
reset	Reset the current synthdef configuration to the self.definition value.
set_context	Set context in SynthDef.
set_contexts	Set multiple values at onces when you give a dictionary.
unset_remaining	This method will remove all existing placeholders in the current def.
add	This method will add the current_def to SuperCollider.s
free	Free this SynthDef from the server.
__repr__	Return repr(self).

classmethod **get_description**(*name: str*, *lang: Optional[sc3nb.sclang.SCLang] = None*) → Optional[Dict[str, *sc3nb.sclang.SynthArgument*]]

Get Synth description

Parameters

name
[str] name of SynthDef

Returns

Dict
dict with SynthArguments

classmethod **send**(*synthdef_bytes: bytes*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

Send a SynthDef as bytes.

Parameters

synthdef_bytes
[bytes] SynthDef bytes

wait
[bool] If True wait for server reply.

server
[SCServer, optional] Server instance that gets the SynthDefs, by default use the SC default server

`classmethod load`(*synthdef_path: str*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

Load SynthDef file at path.

Parameters

synthdef_path
[str] Path with the SynthDefs

server
[SCServer, optional] Server that gets the SynthDefs, by default use the SC default server

`classmethod load_dir`(*synthdef_dir: Optional[str] = None*, *completion_msg: Optional[bytes] = None*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

Load all SynthDefs from directory.

Parameters

synthdef_dir
[str, optional] directory with SynthDefs, by default sc3nb default SynthDefs

completion_msg
[bytes, optional] Message to be executed by the server when loaded, by default None

server
[SCServer, optional] Server that gets the SynthDefs, by default use the SC default server

`reset`() → *SynthDef*

Reset the current synthdef configuration to the self.definition value.

After this you can restart your configuration with the same root definition

Returns

object of type SynthDef
the SynthDef object

`set_context`(*searchpattern: str*, *value*) → *SynthDef*

Set context in SynthDef.

This method will replace a given key (format: "...{{key}}...") in the synthdef definition with the given value.

Parameters

searchpattern
[string] search pattern in the current_def string

value
[string or something with can parsed to string] Replacement of search pattern

Returns

self
[object of type SynthDef] the SynthDef object

`set_contexts`(*dictionary: Dict[str, Any]*) → *SynthDef*

Set multiple values at onces when you give a dictionary.

Because dictionaries are unsorted, keep in mind, that the order is sometimes ignored in this method.

Parameters

dictionary
[dict] {searchpattern: replacement}

Returns

self
[object of type SynthDef] the SynthDef object

`unset_remaining`() → *SynthDef*

This method will remove all existing placeholders in the current def.

You can use this at the end of definition to make sure, that your definition is clean. Hint: This method will not remove pyvars

Returns

self
[object of type SynthDef] the SynthDef object

`add`(*pyvars=None*, *name: Optional[str] = None*, *server: Optional[*sc3nb.sc_objects.server.SCServer*] = None*) → str

This method will add the current_def to SuperCollider.s

If a synth with the same definition was already in sc, this method will only return the name.

Parameters

pyvars
[dict] SC pyvars dict, to inject python variables

name
[str, optional] name which this SynthDef will get

server
[SCServer, optional] Server where this SynthDef will be send to, by default use the SC default server

Returns

str
Name of the SynthDef

`free`() → *SynthDef*

Free this SynthDef from the server.

Returns

self
[object of type SynthDef] the SynthDef object

`__repr__`()

Return repr(self).

class sc3nb.**Buffer**(*bufnum: Optional[int] = None*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

A Buffer object represents a SuperCollider3 Buffer on scsynth and provides access to low-level buffer commands of scsynth via methods of the Buffer objects.

The constructor merely initializes a buffer:

- it selects a buffer number using the server's buffer allocator
- it initializes attribute variables

Parameters

bufnum
[int, optional] buffer number to be used on scsynth. Defaults to None, can be set to enforce a given bufnum

server
[SCServer, optional] The server instance to establish the Buffer, by default use the SC default server

Notes

For more information on Buffer commands, refer to the Server Command Reference in SC3. https://doc.sccode.org/Reference/Server-Command-Reference.html#Buffer%20Commands

Examples

(see examples/buffer-examples.ipynb)

```
>>> b = Buffer().read(...)
>>> b = Buffer().load_data(...)
>>> b = Buffer().alloc(...)
>>> b = Buffer().load_asig(...)
>>> b = Buffer().use_existing(...)
>>> b = Buffer().copy(Buffer)
```

Attributes

server
[the SCServer object] to communicate with scsynth

_bufnum
[int] buffer number = bufnum id on scsynth

_sr
[int] the sampling rate of the buffer

_channels
[int] number of channels of the buffer

_samples
[int] buffer length = number of sample frames

_alloc_mode
[str] ['file', 'alloc', 'data', 'existing', 'copy'] according to previously used generator, defaults to None

_allocated
[boolean] True if Buffer has been allocated by any of the initialization methods

_path
[str] path to the audio file used in load_file()

Overview:

read	Allocate buffer memory and read a sound file.
alloc	Allocate buffer memory.
load_data	Allocate buffer memory and read input data.
load_collection	Wrapper method of *Buffer.load_data()*
load_asig	Create buffer from asig
use_existing	Creates a buffer object from already existing Buffer bufnum.
copy_existing	Duplicate an existing buffer
fill	Fill range of samples with value(s).
gen	Call a command to fill a buffer.
zero	Set buffer data to zero.
gen_sine1	Fill the buffer with sine waves & given amplitude
gen_sine2	Fill the buffer with sine waves
gen_sine3	Fill the buffer with sine waves & given a list of
gen_cheby	Fills a buffer with a series of chebyshev polynomials, which can be
gen_copy	Copy samples from the source buffer to the destination buffer
play	Play the Buffer using a Synth
write	Write buffer data to a sound file
close	Close soundfile after using a Buffer with DiskOut
to_array	Return the buffer data as an array representation.
query	Get buffer info.
__repr__	Return repr(self).
free	Free buffer data.
_gen_flags	Generate Wave Fill Commands flags from booleans

read(*path: str*, *starting_frame: int = 0*, *num_frames: int = -1*, *channels: Optional[Union[int, Sequence[int]]] = None*) → *Buffer*

Allocate buffer memory and read a sound file.

If the number of frames argument num_frames is negative or zero, the entire file is read.

Parameters

path
[string] path name of a sound file.

starting_frame
[int] starting frame in file

num_frames
[int] number of frames to read

channels
[list | int] channels and order of channels to be read from file. if only a int is provided it is loaded as only channel

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is already allocated.

`alloc`(*size: int*, *sr: int = 44100*, *channels: int = 1*) → *Buffer*

Allocate buffer memory.

Parameters

size
[int] number of frames

sr
[int] sampling rate in Hz (optional. default = 44100)

channels
[int] number of channels (optional. default = 1 channel)

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is already allocated.

`load_data`(*data: numpy.ndarray*, *sr: int = 44100*, *mode: str = 'file'*, *sync: bool = True*) → *Buffer*

Allocate buffer memory and read input data.

Parameters

data
[numpy array] Data which should inserted

sr
[int, default: 44100] sample rate

mode
['file' or 'osc'] Insert data via filemode ('file') or n_set OSC commands ('osc') Bundling is only supported for 'osc' mode and if sync is False.

sync: bool, default: True
Use SCServer.sync after sending messages when mode = 'osc'

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is already allocated.

`load_collection`(*data: numpy.ndarray*, *mode: str = 'file'*, *sr: int = 44100*) → *Buffer*

Wrapper method of *Buffer.load_data()*

load_asig(*asig: pya.Asig*, *mode: str = 'file'*) → *Buffer*

Create buffer from asig

Parameters

asig
[pya.Asig] asig to be loaded in buffer

mode
[str, optional] Insert data via filemode ('file') or n_set OSC commands ('osc'), by default 'file'

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is already allocated.

use_existing(*bufnum: int*, *sr: int = 44100*) → *Buffer*

Creates a buffer object from already existing Buffer bufnum.

Parameters

bufnum
[int] buffer node id

sr
[int] Sample rate

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is already allocated.

copy_existing(*buffer:* Buffer) → *Buffer*

Duplicate an existing buffer

Parameters

buffer
[Buffer object] Buffer which should be duplicated

Returns

self
[Buffer] the newly created Buffer object

Raises

RuntimeError
If the Buffer is already allocated.

fill(*start: int = 0*, *count: int = 0*, *value: float = 0*) → *Buffer*

Fill range of samples with value(s).

Parameters

start
[int or list] int : sample starting index list : n*[start, count, value] list

count
[int] number of samples to fill

value
[float] value

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

`gen`(*command: str*, *args: List[Any]*) → *Buffer*

Call a command to fill a buffer. If you know, what you do -> you can use this method.

Parameters

command
[str] What fill command to use.

args
[List[Any]] Arguments for command

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

See also:

gen_sine1, *gen_sine2*, *gen_cheby*, *gen_cheby*, *gen_copy*

`zero`() → *Buffer*

Set buffer data to zero.

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

`gen_sine1`(*amplitudes: List[float]*, *normalize: bool = False*, *wavetable: bool = False*, *clear: bool = False*) → *Buffer*

Fill the buffer with sine waves & given amplitude

Parameters

amplitudes
[list] The first float value specifies the amplitude of the first partial, the second float value specifies the amplitude of the second partial, and so on.

normalize
[bool] Normalize peak amplitude of wave to 1.0.

wavetable
[bool] If set, then the buffer is written in wavetable format so that it can be read by interpolating oscillators.

clear
[bool] If set then the buffer is cleared before new partials are written into it. Otherwise the new partials are summed with the existing contents of the buffer

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

`gen_sine2`(*freq_amps: List[float]*, *normalize: bool = False*, *wavetable: bool = False*, *clear: bool = False*) → *Buffer*

Fill the buffer with sine waves given list of [frequency, amplitude] lists

Parameters

freq_amps
[list] Similar to sine1 except that each partial frequency is specified explicitly instead of being an integer multiple of the fundamental. Non-integer partial frequencies are possible.

normalize
[bool] If set, normalize peak amplitude of wave to 1.0.

wavetable
[bool] If set, the buffer is written in wavetable format so that it can be read by interpolating oscillators.

clear
[bool] If set, the buffer is cleared before new partials are written into it. Otherwise the new partials are summed with the existing contents of the buffer.

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

`gen_sine3`(*freqs_amps_phases: List[float]*, *normalize: bool = False*, *wavetable: bool = False*, *clear: bool = False*) → *Buffer*

Fill the buffer with sine waves & given a list of [frequency, amplitude, phase] entries.

Parameters

freqs_amps_phases
[list] Similar to sine2 except that each partial may have a nonzero starting phase.

normalize
[bool] if set, normalize peak amplitude of wave to 1.0.

wavetable
[bool] If set, the buffer is written in wavetable format so that it can be read by interpolating oscillators.

clear
[bool] If set, the buffer is cleared before new partials are written into it. Otherwise the new partials are summed with the existing contents of the buffer.

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

`gen_cheby`(*amplitudes: List[float]*, *normalize: bool = False*, *wavetable: bool = False*, *clear: bool = False*) → *Buffer*

Fills a buffer with a series of chebyshev polynomials, which can be defined as cheby(n) = amplitude * cos(n * acos(x))

Parameters

amplitudes
[list] The first float value specifies the amplitude for n = 1, the second float value specifies the amplitude for n = 2, and so on

normalize
[bool] If set, normalize the peak amplitude of the Buffer to 1.0.

wavetable
[bool] If set, the buffer is written in wavetable format so that it can be read by interpolating oscillators.

clear
[bool] If set the buffer is cleared before new partials are written into it. Otherwise the new partials are summed with the existing contents of the buffer.

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

`gen_copy`(*source:* Buffer, *source_pos: int*, *dest_pos: int*, *copy_amount: int*) → *Buffer*

Copy samples from the source buffer to the destination buffer specified in the b_gen command.

Parameters

source
[Buffer] Source buffer object

source_pos
[int] sample position in source

dest_pos
[int] sample position in destination

copy_amount
[int] number of samples to copy. If the number of samples to copy is negative, the maximum number of samples possible is copied.

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

`play`(*rate: float = 1*, *loop: bool = False*, *pan: float = 0*, *amp: float = 0.3*) → *sc3nb.sc_objects.node.Synth*

Play the Buffer using a Synth

Parameters

rate
[float, optional] plackback rate, by default 1

loop
[bool, optional] if True loop the playback, by default False

pan
[int, optional] pan position, -1 is left, +1 is right, by default 0

amp
[float, optional] amplitude, by default 0.3

Returns

Synth
Synth to control playback.

Raises

RuntimeError
If the Buffer is not allocated yet.

`write`(*path: str*, *header: str = 'wav'*, *sample: str = 'float'*, *num_frames: int = -1*, *starting_frame: int = 0*, *leave_open: bool = False*) → *Buffer*

Write buffer data to a sound file

Parameters

path
[string] path name of a sound file.

header
[string] header format. Header format is one of: “aiff”, “next”, “wav”, “ircam””, “raw”

sample
[string] sample format. Sample format is one of: “int8”, “int16”, “int24”, “int32”, “float”, “double”, “mulaw”, “alaw”

num_frames
[int] number of frames to write. -1 means all frames.

starting_frame
[int] starting frame in buffer

leave_open
[boolean] Whether you want the buffer file left open. For use with DiskOut you will want this to be true. The file is created, but no frames are written until the DiskOut UGen does so. The default is false which is the correct value for all other cases.

Returns

self
[Buffer] the Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

`close()` → *Buffer*

Close soundfile after using a Buffer with DiskOut

Returns

self
[Buffer] the Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

`to_array()` → numpy.ndarray

Return the buffer data as an array representation.

Returns

np.ndarray:
Values of the buffer

Raises

RuntimeError
If the Buffer is not allocated yet.

`query()` → *BufferInfo*

Get buffer info.

Returns

Tuple:
(buffer number, number of frames, number of channels, sampling rate)

Raises

RuntimeError
If the Buffer is not allocated yet.

`__repr__()` → str

Return repr(self).

free() → None

Free buffer data.

Raises

RuntimeError
If the Buffer is not allocated yet.

_gen_flags(*a_normalize=False*, *a_wavetable=False*, *a_clear=False*) → int

Generate Wave Fill Commands flags from booleans according to the SuperCollider Server Command Reference.

Parameters

a_normalize
[bool, optional] Normalize peak amplitude of wave to 1.0, by default False

a_wavetable
[bool, optional] If set, then the buffer is written in wavetable format so that it can be read by interpolating oscillators, by default False

a_clear
[bool, optional] If set then the buffer is cleared before new partials are written into it. Otherwise the new partials are summed with the existing contents of the buffer, by default False

Returns

int
Wave Fill Commands flags

class sc3nb.**Bus**(*rate: Union[*BusRate*, str]*, *num_channels: int = 1*, *index: Optional[int] = None*, *server: Optional[*sc3nb.sc_objects.server.SCServer*] = None*)

Represenation of Control or Audio Bus(es) on the SuperCollider Server

If num_channels > 1 this will be represent muliple Buses in a row.

Parameters

rate
[Union[BusRate, str]] Rate of the Bus, either control or audio

num_channels
[int, optional] How many channels to allocate, by default 1

index
[int, optional] Starting Bus index this Bus, by default this will be handled by the servers Bus allocator.

server
[SCServer, optional] Server instance for this Bus, by default the default SC server instance.

Overview:

is_audio_bus	Rate check
is_control_bus	Rate check
set	Set ranges of bus values.
fill	Fill bus(es) to one value.
get	Get bus value(s).
free	Mark this Buses ids as free again
__del__	
__repr__	Return repr(self).

is_audio_bus() → bool

Rate check

Returns

bool
True if this is a audio bus

is_control_bus() → bool

Rate check

Returns

bool
True if this is a control bus

set(**values: Sequence[Union[int, float]]*, *return_msg=False*) → Union[*Bus*, *sc3nb.osc.osc_communication.OSCMessage*]

Set ranges of bus values.

Parameters

values
[sequence of int or float] Values that should be set

return_msg
[bool, optional] If True return msg else send it directly, by default False

Raises

RuntimeError
If trying to setn an Audio Bus

fill(*value: Union[int, float]*, *return_msg=False*) → Union[*Bus*, *sc3nb.osc.osc_communication.OSCMessage*]

Fill bus(es) to one value.

Parameters

value
[Union[int, float]] value for the buses

return_msg
[bool, optional] If True return msg else send it directly, by default False

Raises

RuntimeError
If fill is used on a Audio Bus

get() → Union[Union[int, float], Sequence[Union[int, float]]]

Get bus value(s).

Returns

bus value or sequence of bus values
The current value of this bus Multiple values if this bus has num_channels > 1

Raises

RuntimeError
If get is used on an Audio Bus

free(*clear: bool = True*) → None

Mark this Buses ids as free again

Parameters

clear
[bool, optional] Reset bus value(s) to 0, by default True

__del__() → None

__repr__() → str

Return repr(self).

class sc3nb.**Score**

Overview:

load_file	Load a OSC file into a dict.
write_file	Write this score as binary OSC file for NRT synthesis.
record_nrt	Write an OSC file from the messages and wri

classmethod load_file(*path: Union[str, bytes, os.PathLike]*) → Dict[float, List[*sc3nb.osc.osc_communication.OSCMessage*]]

Load a OSC file into a dict.

Parameters

path
[Union[str, bytes, os.PathLike]] Path of the OSC file.

Returns

Dict[float, List[OSCMessage]]
dict with time tag as keys and lists of OSCMessages as values.

classmethod write_file(*messages: Dict[float, List[*sc3nb.osc.osc_communication.OSCMessage*]], path: Union[str, bytes, os.PathLike], tempo: float = 1*)

Write this score as binary OSC file for NRT synthesis.

Parameters

messages
[Dict[float, List[OSCMessage]]] Dict with times as key and lists of OSC messages as values

path
[Union[str, bytes, os.PathLike]] output path for the binary OSC file

tempo
[float] Times will be multiplied by 1/tempo

classmethod record_nrt(*messages: Dict[float, List[sc3nb.osc.osc_communication.OSCMessage]]*, *osc_path: str*, *out_file: str*, *in_file: Optional[str] = None*, *sample_rate: int = 44100*, *header_format: str = 'AIFF'*, *sample_format: str = 'int16'*, *options: Optional[sc3nb.sc_objects.server.ServerOptions] = None*)

Write an OSC file from the messages and wri

Parameters

messages
[Dict[float, List[OSCMessage]]] Dict with times as key and lists of OSC messages as values.

osc_path
[str] Path of the binary OSC file.

out_file
[str] Path of the resulting sound file.

in_file
[Optional[str], optional] Path of input soundfile, by default None.

sample_rate
[int, optional] sample rate for synthesis, by default 44100.

header_format
[str, optional] header format of the output file, by default "AIFF".

sample_format
[str, optional] sample format of the output file, by default "int16".

options
[Optional[ServerOptions], optional] instance of server options to specify server options, by default None

Returns

subprocess.CompletedProcess
Completed scsynth non-realtime process.

class sc3nb.**Recorder**(*path: str = 'record.wav'*, *nr_channels: int = 2*, *rec_header: str = 'wav'*, *rec_format: str = 'int16'*, *bufsize: int = 65536*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

Allows to record audio easily.

Create and prepare a recorder.

Parameters

path
[str, optional] path of recording file, by default "record.wav"

nr_channels
[int, optional] Number of channels, by default 2

rec_header
[str, optional] File format, by default "wav"

rec_format
[str, optional] Recording resolution, by default "int16"

bufsize
[int, optional] size of buffer, by default 65536

server
[SCServer, optional] server used for recording, by default use the SC default server

Overview:

prepare	Pepare the recorder.
start	Start the recording.
pause	Pause the recording.
resume	Resume the recording
stop	Stop the recording.
__repr__	Return repr(self).
__del__	

prepare(*path: str = 'record.wav'*, *nr_channels: int = 2*, *rec_header: str = 'wav'*, *rec_format: str = 'int16'*, *bufsize: int = 65536*)

Pepare the recorder.

Parameters

path
[str, optional] path of recording file, by default "record.wav"

nr_channels
[int, optional] Number of channels, by default 2

rec_header
[str, optional] File format, by default "wav"

rec_format
[str, optional] Recording resolution, by default "int16"

bufsize
[int, optional] size of buffer, by default 65536

Raises

RuntimeError
When Recorder does not needs to be prepared.

start(*timetag: float = 0*, *duration: Optional[float] = None*, *node: Union[*sc3nb.sc_objects.node.Node, *int] = 0*, *bus: int = 0*)

Start the recording.

Parameters

timetag
[float, by default 0 (immediately)] Time (or time offset when <1e6) to start

duration
[float, optional] Length of the recording, by default until stopped.

node
[Union[Node, int], optional] Node that should be recorded, by default 0

bus
[int, by default 0] Bus that should be recorded

Raises

RuntimeError
When trying to start a recording unprepared.

`pause`(*timetag: float = 0*)

Pause the recording.

Parameters

timetag
[float, by default 0 (immediately)] Time (or time offset when <1e6) to pause

Raises

RuntimeError
When trying to pause if not recording.

`resume`(*timetag: float = 0*)

Resume the recording

Parameters

timetag
[float, by default 0 (immediately)] Time (or time offset when <1e6) to resume

Raises

RuntimeError
When trying to resume if not paused.

`stop`(*timetag: float = 0*)

Stop the recording.

Parameters

timetag
[float, by default 0 (immediately)] Time (or time offset when <1e6) to stop

Raises

RuntimeError
When trying to stop if not started.

`__repr__`() → str

Return repr(self).

`__del__`()

`class sc3nb.TimedQueue`(*relative_time: bool = False*, *thread_sleep_time: float = 0.001*, *drop_time_threshold: float = 0.5*)

Accumulates events as timestamps and functions.

Executes given functions according to the timestamps

Parameters

relative_time
[bool, optional] If True, use relative time, by default False

thread_sleep_time
[float, optional] Sleep time in seconds for worker thread, by default 0.001

drop_time_threshold
[float, optional] Threshold for execution time of events in seconds. If this is exceeded the event will be dropped, by default 0.5

Overview:

close	Closes event processing without waiting for pending events
join	Closes event processing after waiting for pending events
complete	Blocks until all pending events have completed
put	Adds event to queue
get	Get latest event from queue and remove event
peek	Look up latest event from queue
empty	Checks if queue is empty
pop	Removes latest event from queue
__worker	Worker function to process events
__repr__	Return repr(self).
elapse	Add time delta to the current queue time.

close() → None

Closes event processing without waiting for pending events

join() → None

Closes event processing after waiting for pending events

complete() → None

Blocks until all pending events have completed

put(*timestamp: float*, *function: Callable[Ellipsis, None]*, *args: Iterable[Any] = ()*, *spawn: bool = False*) → None

Adds event to queue

Parameters

timestamp
[float] Time (POSIX) when event should be executed

function
[Callable[..., None]] Function to be executed

args
[Iterable[Any], optional] Arguments to be passed to function, by default ()

spawn
[bool, optional] if True, create new sub-thread for function, by default False

Raises

TypeError
raised if function is not callable

get() → *Event*

Get latest event from queue and remove event

Returns

Event
Latest event

peek() → *Event*

Look up latest event from queue

Returns

Event
Latest event

`empty()` → bool

Checks if queue is empty

Returns

bool
True if queue if empty

`pop()` → None

Removes latest event from queue

`__worker`(*sleep_time: float*, *close_event: threading.Event*) → NoReturn

Worker function to process events

`__repr__()`

Return repr(self).

`elapse`(*time_delta: float*) → None

Add time delta to the current queue time.

Parameters

time_delta
[float] Additional time

`class sc3nb.TimedQueueSC`(*server: sc3nb.osc.osc_communication.OSCCommunication = None*, *relative_time: bool = False*, *thread_sleep_time: float = 0.001*)

Bases: *TimedQueue*

Timed queue with OSC communication.

Parameters

server
[OSCCommunication, optional] OSC server to handle the bundlers and messsages, by default None

relative_time
[bool, optional] If True, use relative time, by default False

thread_sleep_time
[float, optional] Sleep time in seconds for worker thread, by default 0.001

Overview:

put_bundler	Add a Bundler to queue
put_msg	Add a message to queue

`put_bundler`(*onset: float*, *bundler: sc3nb.osc.osc_communication.Bundler*) → None

Add a Bundler to queue

Parameters

onset
[float] Sending timetag of the Bundler

bundler
[Bundler] Bundler that will be sent

put_msg(*onset: float*, *msg: Union[sc3nb.osc.osc_communication.OSCMessage, str]*, *msg_params: Iterable[Any]*) → None

Add a message to queue

Parameters

onset
[float] Sending timetag of the message

msg
[Union[OSCMessage, str]] OSCMessage or OSC address

msg_params
[Iterable[Any]] If msg is str, this will be the parameters of the created OSCMessage

class sc3nb.**OSCMessage**(*msg_address: str*, *msg_parameters: Optional[Union[Sequence, Any]] = None*)

Class for creating messages to send over OSC

Parameters

msg_address
[str] OSC message address

msg_parameters
[Optional[Union[Sequence]], optional] OSC message parameters, by default None

Overview:

to_pythonosc	Return python-osc OscMessage
_build_message	Builds pythonsosc OSC message.
__repr__	Return repr(self).

to_pythonosc() → pythonosc.osc_message.OscMessage

Return python-osc OscMessage

static _build_message(*msg_address: str*, *msg_parameters: Optional[Union[Sequence, Any]] = None*) → pythonosc.osc_message.OscMessage

Builds pythonsosc OSC message.

Parameters

msg_address
[str] SuperCollider address.

msg_parameters
[list, optional] List of parameters to add to message.

Returns

OscMessage
Message ready to be sent.

__repr__() → str

Return repr(self).

class sc3nb.**Bundler**(*timetag: float = 0*, *msg: Optional[Union[OSCMessage, str]] = None*, *msg_params: Optional[Sequence[Any]] = None*, ***, *server: Optional[OSCCommunication] = None*, *receiver: Optional[Union[str, Tuple[str, int]]] = None*, *send_on_exit: bool = True*)

Class for creating OSCBundles and bundling of messages

Create a Bundler

Parameters

timetag
[float, optional] Starting time at which bundle content should be executed. If timetag > 1e6 it is interpreted as POSIX time. If timetag <= 1e6 it is assumed to be relative value in seconds and is added to time.time(), by default 0, i.e. 'now'.

msg
[OSCMessage or str, optional] OSCMessage or message address, by default None

msg_params
[sequence of any type, optional] Parameters for the message, by default None

server
[OSCCommunication, optional] OSC server, by default None

receiver
[Union[str, Tuple[str, int]], optional] Where to send the bundle, by default send to default receiver of server

send_on_exit
[bool, optional] Whether the bundle is sent when using as context manager, by default True

Overview:

wait	Add time to internal time
add	Add content to this Bundler.
messages	Generate a dict with all messages in this Bundler.
send	Send this Bundler.
to_raw_osc	Create a raw OSC Bundle from this bundler.
to_pythonosc	Build this bundle.
_calc_timetag	
__deepcopy__	
__enter__	
__exit__	
__repr__	Return repr(self).

wait(*time_passed: float*) → None

Add time to internal time

Parameters

time_passed
[float] How much seconds should be passed.

add(**args*) → *Bundler*

Add content to this Bundler.

Parameters

args
[accepts an OSCMessage or Bundler] or a timetag with an OSCMessage or Bundler or Bundler arguments like

(timetag, msg_addr, msg_params) (timetag, msg_addr) (timetag, msg)

Returns

Bundler
self for chaining

messages(*start_time: Optional[float] = 0.0*, *delay: Optional[float] = None*) → Dict[float, List[*OSCMessage*]]

Generate a dict with all messages in this Bundler.

They dict key is the time tag of the messages.

Parameters

start_time
[Optional[float], optional] start time when using relative timing, by default 0.0

Returns

Dict[float, List[OSCMessage]]
dict containg all OSCMessages

send(*server: Optional[OSCCommunication] = None*, *receiver: Tuple[str, int] = None*, *bundle: bool = True*)

Send this Bundler.

Parameters

server
[OSCCommunication, optional] Server instance for sending the bundle. If None it will use the server from init or try to use sc3nb.SC.get_default().server, by default None

receiver
[Tuple[str, int], optional] Address (ip, port) to send to, if None it will send the bundle to the default receiver of the Bundler

bundle
[bool, optional] If True this is allowed to be bundled, by default True

Raises

RuntimeError
When no server could be found.

to_raw_osc(*start_time: Optional[float] = None*, *delay: Optional[float] = None*) → bytes

Create a raw OSC Bundle from this bundler.

Parameters

start_time
[Optional[float], optional] used as start time when using relative timing, by default time.time()

delay: float, optinal
used to delay the timing.

Returns

OscBundle
bundle instance for sending

to_pythonosc(*start_time: Optional[float] = None*, *delay: Optional[float] = None*) → pythonosc.osc_bundle.OscBundle

Build this bundle.

Parameters

start_time
[Optional[float], optional] used as start time when using relative timing, by default time.time()

delay: float, optinal
used to delay the timing.

Returns

OscBundle
bundle instance for sending

_calc_timetag(*start_time: Optional[float]*)

__deepcopy__(*memo*) → *Bundler*

__enter__()

__exit__(*exc_type*, *exc_value*, *exc_traceback*)

__repr__() → str

Return repr(self).

12.2 Subpackages

12.2.1 sc3nb.osc

Module for Open Sound Control (OSC) related code.

12.2.1.1 Submodules

sc3nb.osc.osc_communication

OSC communication

Classes and functions to communicate with SuperCollider using the Open Sound Control (OSC) protocol over UDP

Module Contents

Function List

get_max_udp_packet_size	Get the max UDP packet size by trial and error
split_into_max_size	
convert_to_sc3nb_osc	Get binary OSC representation

Class List

OSCMessage	Class for creating messages to send over OSC
Bundler	Class for creating OSCBundles and bundling of messages
MessageHandler	Base class for Message Handling
MessageQueue	Queue to retrieve OSC messages send to the corresponding OSC address
MessageQueueCollection	A collection of MessageQueues that are all sent to one and the same first address.
OSCCommunication	Class to send and receive OSC messages and bundles.

Content

sc3nb.osc.osc_communication.**_LOGGER**

sc3nb.osc.osc_communication.**get_max_udp_packet_size**()

Get the max UDP packet size by trial and error

sc3nb.osc.osc_communication.**split_into_max_size**(*bundler:* Bundler, *max_dgram_size*) → List[pythonosc.osc_bundle.OscBundle]

class sc3nb.osc.osc_communication.**OSCMessage**(*msg_address: str*, *msg_parameters: Optional[Union[Sequence, Any]] = None*)

Class for creating messages to send over OSC

Parameters

msg_address
[str] OSC message address

msg_parameters
[Optional[Union[Sequence]], optional] OSC message parameters, by default None

Overview:

to_pythonosc	Return python-osc OscMessage
_build_message	Builds pythonsosc OSC message.
__repr__	Return repr(self).

to_pythonosc() → pythonosc.osc_message.OscMessage

Return python-osc OscMessage

static _build_message(*msg_address: str*, *msg_parameters: Optional[Union[Sequence, Any]] = None*) → pythonosc.osc_message.OscMessage

Builds pythonsosc OSC message.

Parameters

msg_address
[str] SuperCollider address.

msg_parameters
[list, optional] List of parameters to add to message.

Returns

OscMessage
Message ready to be sent.

`__repr__`() → str

Return repr(self).

class sc3nb.osc.osc_communication.**Bundler**(*timetag: float = 0*, *msg: Optional[Union[OSCMessage, str]] = None*, *msg_params: Optional[Sequence[Any]] = None*, ***, *server: Optional[OSCCommunication] = None*, *receiver: Optional[Union[str, Tuple[str, int]]] = None*, *send_on_exit: bool = True*)

Class for creating OSCBundles and bundling of messages

Create a Bundler

Parameters

timetag
[float, optional] Starting time at which bundle content should be executed. If timetag > 1e6 it is interpreted as POSIX time. If timetag <= 1e6 it is assumed to be relative value in seconds and is added to time.time(), by default 0, i.e. 'now'.

msg
[OSCMessage or str, optional] OSCMessage or message address, by default None

msg_params
[sequence of any type, optional] Parameters for the message, by default None

server
[OSCCommunication, optional] OSC server, by default None

receiver
[Union[str, Tuple[str, int]], optional] Where to send the bundle, by default send to default receiver of server

send_on_exit
[bool, optional] Whether the bundle is sent when using as context manager, by default True

Overview:

wait	Add time to internal time
add	Add content to this Bundler.
messages	Generate a dict with all messages in this Bundler.
send	Send this Bundler.
to_raw_osc	Create a raw OSC Bundle from this bundler.
to_pythonosc	Build this bundle.
_calc_timetag	
__deepcopy__	
__enter__	
__exit__	
__repr__	Return repr(self).

wait(*time_passed: float*) → None

Add time to internal time

Parameters

time_passed
[float] How much seconds should be passed.

add(**args*) → *Bundler*

Add content to this Bundler.

Parameters

args
[accepts an OSCMessage or Bundler] or a timetag with an OSCMessage or Bundler or Bundler arguments like

(timetag, msg_addr, msg_params) (timetag, msg_addr) (timetag, msg)

Returns

Bundler
self for chaining

messages(*start_time: Optional[float] = 0.0*, *delay: Optional[float] = None*) → Dict[float, List[*OSCMessage*]]

Generate a dict with all messages in this Bundler.

They dict key is the time tag of the messages.

Parameters

start_time
[Optional[float], optional] start time when using relative timing, by default 0.0

Returns

Dict[float, List[OSCMessage]]
dict containg all OSCMessages

send(*server: Optional[OSCCommunication] = None*, *receiver: Tuple[str, int] = None*, *bundle: bool = True*)

Send this Bundler.

Parameters

server
[OSCCommunication, optional] Server instance for sending the bundle. If None it will use the server from init or try to use sc3nb.SC.get_default().server, by default None

receiver
[Tuple[str, int], optional] Address (ip, port) to send to, if None it will send the bundle to the default receiver of the Bundler

bundle
[bool, optional] If True this is allowed to be bundled, by default True

Raises

RuntimeError
When no server could be found.

to_raw_osc(*start_time: Optional[float] = None*, *delay: Optional[float] = None*) → bytes

Create a raw OSC Bundle from this bundler.

Parameters

start_time
[Optional[float], optional] used as start time when using relative timing, by default time.time()

delay: float, optinal
used to delay the timing.

Returns

OscBundle
bundle instance for sending

`to_pythonosc`(*start_time: Optional[float] = None*, *delay: Optional[float] = None*) → pythonosc.osc_bundle.OscBundle

Build this bundle.

Parameters

start_time
[Optional[float], optional] used as start time when using relative timing, by default time.time()

delay: float, optinal
used to delay the timing.

Returns

OscBundle
bundle instance for sending

`_calc_timetag`(*start_time: Optional[float]*)

`__deepcopy__`(*memo*) → *Bundler*

`__enter__`()

`__exit__`(*exc_type*, *exc_value*, *exc_traceback*)

`__repr__`() → str

Return repr(self).

sc3nb.osc.osc_communication.**convert_to_sc3nb_osc**(*data: Union[*OSCMessage, Bundler, *pythonosc.osc_message.OscMessage, pythonosc.osc_bundle.OscBundle, bytes]*) → Union[*OSCMessage*, *Bundler*]

Get binary OSC representation

Parameters

package
[Union[OscMessage, Bundler, OscBundle]] OSC Package object

Returns

bytes
raw OSC binary representation of OSC Package

Raises

ValueError
If package is not supported

class sc3nb.osc.osc_communication.**MessageHandler**

Bases: abc.ABC

Base class for Message Handling

Overview:

put	Add message to MessageHandler

abstract put(*address: str*, **args*) → None

Add message to MessageHandler

Parameters

address
[str] Message address

class sc3nb.osc.osc_communication.**MessageQueue**(*address: str*, *preprocess: Optional[Callable] = None*)

Bases: *MessageHandler*

Queue to retrieve OSC messages send to the corresponding OSC address

Create a new AddressQueue

Parameters

address
[str] OSC address for this queue

preprocess
[function, optional]

function that will be applied to the value before they are enqueued
(Default value = None)

Overview:

put	Add a message to MessageQueue
skipped	Skipp one queue value
get	Returns a value from the queue
show	Print the content of the queue.
*_repr_pretty_*	

put(*address: str*, **args*) → None

Add a message to MessageQueue

Parameters

address
[str] message address

skipped()

Skipp one queue value

get(*timeout: float = 5*, *skip: bool = True*) → Any

Returns a value from the queue

Parameters

timeout
[int, optional] Time in seconds that will be waited on the queue, by default 5

skip
[bool, optional] If True the queue will skip as many values as *skips*, by default True

Returns

obj
value from queue

Raises

Empty
If the queue has no value

show() → None

Print the content of the queue.

_repr_pretty_(*printer*, *cycle*) → None

class sc3nb.osc.osc_communication.**MessageQueueCollection**(*address: str*, *sub_addrs: Optional[Sequence[str]] = None*)

Bases: *MessageHandler*

A collection of MessageQueues that are all sent to one and the same first address.

Create a collection of MessageQueues under the same first address

Parameters

address
[str] first message address that is the same for all MessageQueues

sub_addrs
[Optional[Sequence[str]], optional] secound message addresses with seperate queues, by default None Additional MessageQueues will be created on demand.

Overview:

put	Add a message to the corresponding MessageQueue
__contains__	
__getitem__	

put(*address: str*, **args*) → None

Add a message to the corresponding MessageQueue

Parameters

address
[str] first message address

__contains__(*item*) → bool

__getitem__(*key*)

exception sc3nb.osc.osc_communication.**OSCCommunicationError**(*message*, *send_message*)

Bases: Exception

Exception for OSCCommunication errors.

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

class sc3nb.osc.osc_communication.**OSCCommunication**(*server_ip: str*, *server_port: int*, *default_receiver_ip: str*, *default_receiver_port: int*)

Class to send and receive OSC messages and bundles.

Create an OSC communication server

Parameters

server_ip
[str] IP address to use for this server

server_port
[int] port to use for this server

default_receiver_ip
[str] IP address used for sending by default

default_receiver_port
[int] port used for sending by default

Overview:

add_msg_pairs	Add the provided pairs for message receiving.
add_msg_queue	Add a MessageQueue to this servers dispatcher
add_msg_queue_collection	Add a MessageQueueCollection
_check_sender	
lookup_receiver	Reverse lookup the address of a specific receiver
connection_info	Get information about the known addresses
add_receiver	Adds a receiver with the specified address.
send	Sends OSC packet
_handle_outgoing_message	
get_reply_address	Get the corresponding reply address for the given address
msg	Creates and sends OSC message over UDP.
bundler	Generate a Bundler.
quit	Shuts down the sc3nb OSC server

add_msg_pairs(*msg_pairs: Dict[str, str]*) → None

Add the provided pairs for message receiving.

Parameters

msg_pairs
[dict[str, str], optional] dict containing user specified message pairs. {msg_addr: reply_addr}

add_msg_queue(*msg_queue:* MessageQueue, *out_addr: Optional[str] = None*) → None

Add a MessageQueue to this servers dispatcher

Parameters

msg_queue
[MessageQueue] new MessageQueue

out_addr
[Optional[str], optional] The outgoing message address that belongs to this MessageQeue, by default None

`add_msg_queue_collection`(*msg_queue_collection:* MessageQueueCollection) → None

Add a MessageQueueCollection

Parameters

msg_queue_collection
[MessageQueueCollection] MessageQueueCollection to be added

`_check_sender`(*sender: Tuple[str, int]*) → Union[str, Tuple[str, int]]

`lookup_receiver`(*receiver: Union[str, Tuple[str, int]]*) → Tuple[str, int]

Reverse lookup the address of a specific receiver

Parameters

receiver
[str] Receiver name.

Returns

Tuple[str, int]
Receiver address (ip, port)

Raises

KeyError
If receiver is unknown.

ValueError
If the type of the receiver argument is wrong.

`connection_info`(*print_info: bool = True*) → Tuple[Tuple[str, int], Dict[Tuple[str, int], str]]

Get information about the known addresses

Parameters

print_info
[bool, optional]

If True print connection information
(Default value = True)

Returns

tuple
containing the address of this sc3nb OSC Server and known receivers addresses in a dict with their names as values

`add_receiver`(*name: str*, *ip_address: str*, *port: int*)

Adds a receiver with the specified address.

Parameters

name
[str] Name of receiver.

ip_address
[str] IP address of receiver (e.g. "127.0.0.1")

port
[int] Port of the receiver

send(*package: Union[OSCMessage, Bundler]*, ***, *receiver: Optional[Union[str, Tuple[str, int]]] = None*, *bundle: bool = False*, *await_reply: bool = True*, *timeout: float = 5*) → Any

Sends OSC packet

Parameters

package
[OSCMessage or Bundler] Object with *dgram* attribute.

receiver
[str or Tuple[str, int], optional] Where to send the packet, by default send to default receiver

bundle
[bool, optional] If True it is allowed to bundle the package with bundling, by default False.

await_reply
[bool, optional] If True ask for reply from the server and return it, otherwise send the message and return None directly, by default True. If the package is bundled None will be returned.

timeout
[int, optional] timeout in seconds for reply, by default 5

Returns

None or reply
None if no reply was received or awaited else reply.

Raises

ValueError
When the provided package is not supported.

OSCCommunicationError
When the handling of a package fails.

_handle_outgoing_message(*message: OSCMessage*, *receiver_address: Tuple[str, int]*, *await_reply: bool*, *timeout: float*) → Any

get_reply_address(*msg_address: str*) → Optional[str]

Get the corresponding reply address for the given address

Parameters

msg_address
[str] outgoing message address

Returns

str or None
Corresponding reply address if available

msg(*msg_addr: str*, *msg_params: Optional[Sequence] = None*, ***, *bundle: bool = False*, *receiver: Optional[Tuple[str, int]] = None*, *await_reply: bool = True*, *timeout: float = 5*) → Optional[Any]

Creates and sends OSC message over UDP.

Parameters

msg_addr
[str] SuperCollider address of the OSC message

msg_params
[Optional[Sequence], optional] List of paramters of the OSC message, by default None

bundle
[bool, optional] If True it is allowed to bundle the content with bundling, by default False

receiver
[tuple[str, int], optional] (IP address, port) to send the message, by default send to default receiver

await_reply
[bool, optional] If True send message and wait for reply otherwise send the message and return directly, by default True

timeout
[float, optional] timeout in seconds for reply, by default 5

Returns

obj
reply if await_reply and there is a reply for this

`bundler`(*timetag: float = 0*, *msg: Optional[Union[OSCMessage, str]] = None*, *msg_params: Optional[Sequence[Any]] = None*, *send_on_exit: bool = True*) → *Bundler*

Generate a Bundler.

This allows the user to easly add messages/bundles and send it.

Parameters

timetag
[int] Time at which bundle content should be executed. If timetag <= 1e6 it is added to time.time().

msg
[OSCMessage or str, optional] OSCMessage or message address, by default None

msg_params
[sequence of any type, optional] Parameters for the message, by default None

send_on_exit
[bool, optional] Whether the bundle is sent when using as context manager, by default True

Returns

Bundler
bundler for OSC bundling.

`quit`() → None

Shuts down the sc3nb OSC server

sc3nb.osc.parsing

Module for parsing OSC packets from sclang.

This implements an extension of the OSC protocol. A bundle is now allowed to consist of other bundles or lists.

This extension is needed as sclang is sending Arrays as this list or when nested as bundles with inner list

Module Contents

Function List

_get_aligned_index	Get next multiple of NUM_SIZE from index
_parse_list	Parse a OSC List
_parse_osc_bundle_element	Parse an element from an OSC bundle.
_parse_bundle	Parsing bundle
parse_sclang_osc_packet	Parses the OSC packet from sclang.
preprocess_return	Preprocessing function for /return values

Content

sc3nb.osc.parsing.**_LOGGER**

sc3nb.osc.parsing.**SYNTH_DEF_MARKER = b'SCgf'**

sc3nb.osc.parsing.**TYPE_TAG_MARKER**

sc3nb.osc.parsing.**TYPE_TAG_INDEX = 4**

sc3nb.osc.parsing.**NUM_SIZE = 4**

sc3nb.osc.parsing.**BYTES_2_TYPE**

exception sc3nb.osc.parsing.**ParseError**

Bases: Exception

Base exception for when a datagram parsing error occurs.

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

sc3nb.osc.parsing.**_get_aligned_index**(*index: int*) → int

Get next multiple of NUM_SIZE from index

Parameters

index
[int] starting index

Returns

int
next multiple of NUM_SIZE from index

sc3nb.osc.parsing._parse_list(*dgram: bytes*, *start_index: int*) → Tuple[Sequence[Any], int]

Parse a OSC List

List consists of the following bytes: 4 bytes (int) : list_size n bytes (string) : OSC type tag n bytes (x) : content as specified by type tag

Parameters

dgram
[bytes] datagram with the list

start_index
[int] parsing starting index

Returns

Tuple[Sequence[Any], int]
parsed list contents, starting index + number of consumed bytes

Raises

ParseError
If datagram is invalid.

sc3nb.osc.parsing._parse_osc_bundle_element(*dgram: bytes*, *start_index: int*) → Tuple[Union[Sequence[Any], bytes], int]

Parse an element from an OSC bundle.

The element needs to be either an OSC bundle or a list

Parameters

dgram
[bytes] datagram with the bundle element

start_index
[int] parsing starting index

Returns

Tuple[Union[Sequence[Any], bytes], int]
parsed content of the bundle element, starting index + number of consumed bytes

Raises

ParseError
If the datagram is invalid.

sc3nb.osc.parsing._parse_bundle(*dgram: bytes*, *start_index: int*) → Tuple[Sequence[Any], int]

Parsing bundle

Parameters

dgram
[bytes] datagram with the bundle

start_index
[int] parsing starting index

Returns

tuple[Sequence[Any], int]
parsed content, starting index + number of consumed bytes

Raises

ParseError
If the datagram is invalid

sc3nb.osc.parsing.**parse_sclang_osc_packet**(*data: bytes*) → Union[bytes, Sequence[Any]]

Parses the OSC packet from sclang.

Parameters

data
[bytes] bytes sent by sclang

Returns

bytes or Sequence[Any]
unchanged bytes or content of bundles/messages

sc3nb.osc.parsing.**preprocess_return**(*value: Sequence[Any]*) → Sequence[Any]

Preprocessing function for /return values

Parameters

value
[tuple] return data

Returns

obj
data

12.2.2 sc3nb.resources

Module for resources

12.2.2.1 Subpackages

sc3nb.resources.synthdefs

12.2.3 sc3nb.sc_objects

In this module are all SuperCollider Object related classes

12.2.3.1 Submodules

sc3nb.sc_objects.allocators

Classes for managing ID allocations.

Module Contents

Class List

Allocator	Helper class that provides a standard way to create an ABC using
NodeAllocator	Allows allocating ids for Nodes.
BlockAllocator	Allows allocating blocks of ids / indexes

Content

class sc3nb.sc_objects.allocators.**Allocator**

Bases: abc.ABC

Helper class that provides a standard way to create an ABC using inheritance.

Overview:

allocate
free

abstract allocate(*num: int = 1*) → Sequence[int]

abstract free(*ids: Sequence[int]*) → None

class sc3nb.sc_objects.allocators.**NodeAllocator**(*client_id: int*)

Bases: *Allocator*

Allows allocating ids for Nodes.

Overview:

allocate
free

allocate(*num: int = 1*) → Sequence[int]

free(*ids: Sequence[int]*) → None

class sc3nb.sc_objects.allocators.**BlockAllocator**(*num_ids: int*, *offset: int*)

Bases: *Allocator*

Allows allocating blocks of ids / indexes

Overview:

allocate	Allocate the next free ids
free	Mark ids as free again.

allocate(*num: int = 1*) → Sequence[int]

Allocate the next free ids

Returns

int
free ids

Raises

RuntimeError
When out of free ids or not enough ids are in order.

free(*ids: Sequence[int]*) → None

Mark ids as free again.

Parameters

ids
[sequence of int] ids that are not used anymore.

sc3nb.sc_objects.buffer

Module for using SuperCollider Buffers in Python

Module Contents

Class List

BufferReply	Buffer Command Replies
BufferCommand	Buffer OSC Commands for Buffers
BufferAllocationMode	Buffer Allocation Modes
BufferInfo	Information about the Buffer
Buffer	A Buffer object represents a SuperCollider3 Buffer on scsynth

Content

class sc3nb.sc_objects.buffer.**BufferReply**

Bases: str, enum.Enum

Buffer Command Replies

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

INFO = '/b_info'

class sc3nb.sc_objects.buffer.**BufferCommand**

Bases: str, enum.Enum

Buffer OSC Commands for Buffers

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

ALLOC = '/b_alloc'

ALLOC_READ = '/b_allocRead'

ALLOC_READ_CHANNEL = '/b_allocReadChannel'

READ = '/b_read'

READ_CHANNEL = '/b_readChannel'

WRITE = '/b_write'

FREE = '/b_free'

ZERO = '/b_zero'

SET = '/b_set'

SETN = '/b_setn'

FILL = '/b_fill'

GEN = '/b_gen'

CLOSE = '/b_close'

QUERY = '/b_query'

GET = '/b_get'

GETN = '/b_getn'

class sc3nb.sc_objects.buffer.**BufferAllocationMode**

Bases: str, enum.Enum

Buffer Allocation Modes

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

FILE = 'file'

ALLOC = 'alloc'

DATA = 'data'

EXISTING = 'existing'

COPY = 'copy'

NONE = 'none'

class sc3nb.sc_objects.buffer.**BufferInfo**

Bases: NamedTuple

Information about the Buffer

bufnum: int

num_frames: int

num_channels: int

sample_rate: float

class sc3nb.sc_objects.buffer.**Buffer**(*bufnum: Optional[int] = None*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

A Buffer object represents a SuperCollider3 Buffer on scsynth and provides access to low-level buffer commands of scsynth via methods of the Buffer objects.

The constructor merely initializes a buffer:

- it selects a buffer number using the server's buffer allocator
- it initializes attribute variables

Parameters

bufnum
[int, optional] buffer number to be used on scsynth. Defaults to None, can be set to enforce a given bufnum

server
[SCServer, optional] The server instance to establish the Buffer, by default use the SC default server

Notes

For more information on Buffer commands, refer to the Server Command Reference in SC3. https://doc.sccode.org/Reference/Server-Command-Reference.html#Buffer%20Commands

Examples

(see examples/buffer-examples.ipynb)

```
>>> b = Buffer().read(...)
>>> b = Buffer().load_data(...)
>>> b = Buffer().alloc(...)
>>> b = Buffer().load_asig(...)
>>> b = Buffer().use_existing(...)
>>> b = Buffer().copy(Buffer)
```

Attributes

server
[the SCServer object] to communicate with scsynth

_bufnum
[int] buffer number = bufnum id on scsynth

_sr
[int] the sampling rate of the buffer

_channels
[int] number of channels of the buffer

_samples
[int] buffer length = number of sample frames

_alloc_mode
[str] [‘file’, ‘alloc’, ‘data’, ‘existing’, ‘copy’] according to previously used generator, defaults to None

_allocated
[boolean] True if Buffer has been allocated by any of the initialization methods

_path
[str] path to the audio file used in load_file()

Overview:

read	Allocate buffer memory and read a sound file.
alloc	Allocate buffer memory.
load_data	Allocate buffer memory and read input data.
load_collection	Wrapper method of *Buffer.load_data()*
load_asig	Create buffer from asig
use_existing	Creates a buffer object from already existing Buffer bufnum.
copy_existing	Duplicate an existing buffer
fill	Fill range of samples with value(s).
gen	Call a command to fill a buffer.
zero	Set buffer data to zero.
gen_sine1	Fill the buffer with sine waves & given amplitude
gen_sine2	Fill the buffer with sine waves
gen_sine3	Fill the buffer with sine waves & given a list of
gen_cheby	Fills a buffer with a series of chebyshev polynomials, which can be
gen_copy	Copy samples from the source buffer to the destination buffer
play	Play the Buffer using a Synth
write	Write buffer data to a sound file
close	Close soundfile after using a Buffer with DiskOut
to_array	Return the buffer data as an array representation.
query	Get buffer info.
__repr__	Return repr(self).
free	Free buffer data.
_gen_flags	Generate Wave Fill Commands flags from booleans

read(*path: str*, *starting_frame: int = 0*, *num_frames: int = -1*, *channels: Optional[Union[int, Sequence[int]]] = None*) → *Buffer*

Allocate buffer memory and read a sound file.

If the number of frames argument num_frames is negative or zero, the entire file is read.

Parameters

path
[string] path name of a sound file.

starting_frame
[int] starting frame in file

num_frames
[int] number of frames to read

channels
[list | int] channels and order of channels to be read from file. if only a int is provided it is loaded as only channel

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is already allocated.

`alloc`(*size: int*, *sr: int = 44100*, *channels: int = 1*) → *Buffer*

Allocate buffer memory.

Parameters

size
[int] number of frames

sr
[int] sampling rate in Hz (optional. default = 44100)

channels
[int] number of channels (optional. default = 1 channel)

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is already allocated.

`load_data`(*data: numpy.ndarray*, *sr: int = 44100*, *mode: str = 'file'*, *sync: bool = True*) → *Buffer*

Allocate buffer memory and read input data.

Parameters

data
[numpy array] Data which should inserted

sr
[int, default: 44100] sample rate

mode
[‘file’ or ‘osc’] Insert data via filemode (‘file’) or n_set OSC commands (‘osc’) Bundling is only supported for ‘osc’ mode and if sync is False.

sync: bool, default: True
Use SCServer.sync after sending messages when mode = ‘osc’

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is already allocated.

load_collection(*data: numpy.ndarray*, *mode: str = 'file'*, *sr: int = 44100*) → *Buffer*

Wrapper method of *Buffer.load_data()*

load_asig(*asig: pya.Asig*, *mode: str = 'file'*) → *Buffer*

Create buffer from asig

Parameters

asig
[pya.Asig] asig to be loaded in buffer

mode
[str, optional] Insert data via filemode ('file') or n_set OSC commands ('osc'), by default 'file'

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is already allocated.

use_existing(*bufnum: int*, *sr: int = 44100*) → *Buffer*

Creates a buffer object from already existing Buffer bufnum.

Parameters

bufnum
[int] buffer node id

sr
[int] Sample rate

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is already allocated.

copy_existing(*buffer:* Buffer) → *Buffer*

Duplicate an existing buffer

Parameters

buffer
[Buffer object] Buffer which should be duplicated

Returns

self
[Buffer] the newly created Buffer object

Raises

RuntimeError
If the Buffer is already allocated.

fill(*start: int = 0*, *count: int = 0*, *value: float = 0*) → *Buffer*

Fill range of samples with value(s).

Parameters

start
[int or list] int : sample starting index list : n*[start, count, value] list

count
[int] number of samples to fill

value
[float] value

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

gen(*command: str*, *args: List[Any]*) → *Buffer*

Call a command to fill a buffer. If you know, what you do -> you can use this method.

Parameters

command
[str] What fill command to use.

args
[List[Any]] Arguments for command

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

See also:

gen_sine1, *gen_sine2*, *gen_cheby*, *gen_cheby*, *gen_copy*

zero() → *Buffer*

Set buffer data to zero.

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

gen_sine1(*amplitudes: List[float]*, *normalize: bool = False*, *wavetable: bool = False*, *clear: bool = False*) → *Buffer*

Fill the buffer with sine waves & given amplitude

Parameters

amplitudes
[list] The first float value specifies the amplitude of the first partial, the second float value specifies the amplitude of the second partial, and so on.

normalize
[bool] Normalize peak amplitude of wave to 1.0.

wavetable
[bool] If set, then the buffer is written in wavetable format so that it can be read by interpolating oscillators.

clear
[bool] If set then the buffer is cleared before new partials are written into it. Otherwise the new partials are summed with the existing contents of the buffer

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

gen_sine2(*freq_amps: List[float]*, *normalize: bool = False*, *wavetable: bool = False*, *clear: bool = False*) → *Buffer*

Fill the buffer with sine waves given list of [frequency, amplitude] lists

Parameters

freq_amps
[list] Similar to sine1 except that each partial frequency is specified explicitly instead of being an integer multiple of the fundamental. Non-integer partial frequencies are possible.

normalize
[bool] If set, normalize peak amplitude of wave to 1.0.

wavetable
[bool] If set, the buffer is written in wavetable format so that it can be read by interpolating oscillators.

clear
[bool] If set, the buffer is cleared before new partials are written into it. Otherwise the new partials are summed with the existing contents of the buffer.

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

gen_sine3(*freqs_amps_phases: List[float]*, *normalize: bool = False*, *wavetable: bool = False*, *clear: bool = False*) → *Buffer*

Fill the buffer with sine waves & given a list of [frequency, amplitude, phase] entries.

Parameters

freqs_amps_phases
[list] Similar to sine2 except that each partial may have a nonzero starting phase.

normalize
[bool] if set, normalize peak amplitude of wave to 1.0.

wavetable
[bool] If set, the buffer is written in wavetable format so that it can be read by interpolating oscillators.

clear
[bool] If set, the buffer is cleared before new partials are written into it. Otherwise the new partials are summed with the existing contents of the buffer.

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

gen_cheby(*amplitudes: List[float]*, *normalize: bool = False*, *wavetable: bool = False*, *clear: bool = False*) → *Buffer*

Fills a buffer with a series of chebyshev polynomials, which can be defined as cheby(n) = amplitude * cos(n * acos(x))

Parameters

amplitudes
[list] The first float value specifies the amplitude for n = 1, the second float value specifies the amplitude for n = 2, and so on

normalize
[bool] If set, normalize the peak amplitude of the Buffer to 1.0.

wavetable
[bool] If set, the buffer is written in wavetable format so that it can be read by interpolating oscillators.

clear
[bool] If set the buffer is cleared before new partials are written into it. Otherwise the new partials are summed with the existing contents of the buffer.

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

gen_copy(*source:* Buffer, *source_pos: int*, *dest_pos: int*, *copy_amount: int*) → *Buffer*

Copy samples from the source buffer to the destination buffer specified in the b_gen command.

Parameters

source
[Buffer] Source buffer object

source_pos
[int] sample position in source

dest_pos
[int] sample position in destination

copy_amount
[int] number of samples to copy. If the number of samples to copy is negative, the maximum number of samples possible is copied.

Returns

self
[Buffer] the created Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

play(*rate: float = 1*, *loop: bool = False*, *pan: float = 0*, *amp: float = 0.3*) → *sc3nb.sc_objects.node.Synth*

Play the Buffer using a Synth

Parameters

rate
[float, optional] plackback rate, by default 1

loop
[bool, optional] if True loop the playback, by default False

pan
[int, optional] pan position, -1 is left, +1 is right, by default 0

amp
[float, optional] amplitude, by default 0.3

Returns

Synth
Synth to control playback.

Raises

RuntimeError
If the Buffer is not allocated yet.

write(*path: str*, *header: str = 'wav'*, *sample: str = 'float'*, *num_frames: int = -1*, *starting_frame: int = 0*, *leave_open: bool = False*) → *Buffer*

Write buffer data to a sound file

Parameters

path
[string] path name of a sound file.

header
[string] header format. Header format is one of: “aiff”, “next”, “wav”, “ircam””, “raw”

sample
[string] sample format. Sample format is one of: “int8”, “int16”, “int24”, “int32”, “float”, “double”, “mulaw”, “alaw”

num_frames
[int] number of frames to write. -1 means all frames.

starting_frame
[int] starting frame in buffer

leave_open
[boolean] Whether you want the buffer file left open. For use with DiskOut you will want this to be true. The file is created, but no frames are written until the DiskOut UGen does so. The default is false which is the correct value for all other cases.

Returns

self
[Buffer] the Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

`close()` → *Buffer*

Close soundfile after using a Buffer with DiskOut

Returns

self
[Buffer] the Buffer object

Raises

RuntimeError
If the Buffer is not allocated yet.

`to_array()` → numpy.ndarray

Return the buffer data as an array representation.

Returns

np.ndarray:
Values of the buffer

Raises

RuntimeError
If the Buffer is not allocated yet.

`query()` → *BufferInfo*

Get buffer info.

Returns

Tuple:
(buffer number, number of frames, number of channels, sampling rate)

Raises

RuntimeError
If the Buffer is not allocated yet.

`__repr__`() → str

Return repr(self).

`free`() → None

Free buffer data.

Raises

RuntimeError
If the Buffer is not allocated yet.

`_gen_flags`(*a_normalize=False*, *a_wavetable=False*, *a_clear=False*) → int

Generate Wave Fill Commands flags from booleans according to the SuperCollider Server Command Reference.

Parameters

a_normalize
[bool, optional] Normalize peak amplitude of wave to 1.0, by default False

a_wavetable
[bool, optional] If set, then the buffer is written in wavetable format so that it can be read by interpolating oscillators, by default False

a_clear
[bool, optional] If set then the buffer is cleared before new partials are written into it. Otherwise the new partials are summed with the existing contents of the buffer, by default False

Returns

int
Wave Fill Commands flags

sc3nb.sc_objects.bus

Python representation of the scsynth Bus.

Module Contents

Class List

ControlBusCommand	OSC Commands for Control Buses
BusRate	Calculation rate of Buses
Bus	Represenation of Control or Audio Bus(es) on the SuperCollider Server

Content

class sc3nb.sc_objects.bus.**ControlBusCommand**

Bases: str, enum.Enum

OSC Commands for Control Buses

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

FILL = '/c_fill'

SET = '/c_set'

SETN = '/c_setn'

GET = '/c_get'

GETN = '/c_getn'

class sc3nb.sc_objects.bus.**BusRate**

Bases: str, enum.Enum

Calculation rate of Buses

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

AUDIO = 'audio'

CONTROL = 'control'

class sc3nb.sc_objects.bus.**Bus**(*rate: Union[*BusRate*, str]*, *num_channels: int = 1*, *index: Optional[int] = None*, *server: Optional[*sc3nb.sc_objects.server.SCServer*] = None*)

Represenation of Control or Audio Bus(es) on the SuperCollider Server

If num_channels > 1 this will be represent muliple Buses in a row.

Parameters

rate
[Union[BusRate, str]] Rate of the Bus, either control or audio

num_channels
[int, optional] How many channels to allocate, by default 1

index
[int, optional] Starting Bus index this Bus, by default this will be handled by the servers Bus allocator.

server
[SCServer, optional] Server instance for this Bus, by default the default SC server instance.

Overview:

is_audio_bus	Rate check
is_control_bus	Rate check
set	Set ranges of bus values.
fill	Fill bus(es) to one value.
get	Get bus value(s).
free	Mark this Buses ids as free again
__del__	
__repr__	Return repr(self).

is_audio_bus() → bool

Rate check

Returns

bool
True if this is a audio bus

is_control_bus() → bool

Rate check

Returns

bool
True if this is a control bus

set(**values: Sequence[Union[int, float]]*, *return_msg=False*) → Union[*Bus*, *sc3nb.osc.osc_communication.OSCMessage*]

Set ranges of bus values.

Parameters

values
[sequence of int or float] Values that should be set

return_msg
[bool, optional] If True return msg else send it directly, by default False

Raises

RuntimeError
If trying to setn an Audio Bus

fill(*value: Union[int, float]*, *return_msg=False*) → Union[*Bus*, *sc3nb.osc.osc_communication.OSCMessage*]

Fill bus(es) to one value.

Parameters

value
[Union[int, float]] value for the buses

return_msg
[bool, optional] If True return msg else send it directly, by default False

Raises

RuntimeError
If fill is used on a Audio Bus

get() → Union[Union[int, float], Sequence[Union[int, float]]]

Get bus value(s).

Returns

bus value or sequence of bus values
The current value of this bus Multiple values if this bus has num_channels > 1

Raises

RuntimeError
If get is used on an Audio Bus

free(*clear: bool = True*) → None

Mark this Buses ids as free again

Parameters

clear
[bool, optional] Reset bus value(s) to 0, by default True

__del__() → None

__repr__() → str

Return repr(self).

sc3nb.sc_objects.node

Implements Node and subclasses Synth and Group.

Module Contents

Class List

GroupReply	Replies of Group Commands
GroupCommand	OSC Commands for Groups
SynthCommand	OSC Commands for Synths
NodeReply	Replies of Node Commands
NodeCommand	OSC Commands for Nodes
AddAction	AddAction of SuperCollider nodes.
SynthInfo	Information about the Synth from /n_info
GroupInfo	Information about the Group from /n_info
Node	Representation of a Node on SuperCollider.
Synth	Representation of a Synth on SuperCollider.
Group	Representation of a Group on SuperCollider.
NodeTree	Node Tree is a class for parsing /g_queryTree.reply

Content

sc3nb.sc_objects.node._LOGGER

class sc3nb.sc_objects.node.GroupReply

Bases: str, enum.Enum

Replies of Group Commands

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

QUERY_TREE_REPLY = '/g_queryTree.reply'

class sc3nb.sc_objects.node.GroupCommand

Bases: str, enum.Enum

OSC Commands for Groups

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

QUERY_TREE = '/g_queryTree'

DUMP_TREE = '/g_dumpTree'

DEEP_FREE = '/g_deepFree'

FREE_ALL = '/g_freeAll'

TAIL = '/g_tail'

HEAD = '/g_head'

G_NEW = '/g_new'

P_NEW = '/p_new'

class sc3nb.sc_objects.node.SynthCommand

Bases: str, enum.Enum

OSC Commands for Synths

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

NEW = '/s_new'

S_GET = '/s_get'

S_GETN = '/s_getn'

class sc3nb.sc_objects.node.NodeReply

Bases: str, enum.Enum

Replies of Node Commands

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

INFO = '/n_info'

class sc3nb.sc_objects.node.**NodeCommand**

Bases: str, enum.Enum

OSC Commands for Nodes

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

ORDER = '/n_order'

TRACE = '/n_trace'

QUERY = '/n_query'

MAP = '/n_map'

MAPN = '/n_mapn'

MAPA = '/n_mapa'

MAPAN = '/n_mapan'

FILL = '/n_fill'

SET = '/n_set'

SETN = '/n_setn'

RUN = '/n_run'

FREE = '/n_free'

class sc3nb.sc_objects.node.**AddAction**

Bases: enum.Enum

AddAction of SuperCollider nodes.

This Enum contains the codes for the different ways to add a node.

TO_HEAD = 0

TO_TAIL = 1

BEFORE = 2

AFTER = 3

REPLACE = 4

class sc3nb.sc_objects.node.**SynthInfo**

Bases: NamedTuple

Information about the Synth from /n_info

nodeid: int

group: int

prev_nodeid: int

next_nodeid: int

class sc3nb.sc_objects.node.**GroupInfo**

Bases: NamedTuple

Information about the Group from /n_info

nodeid: int

group: int

prev_nodeid: int

next_nodeid: int

head: int

tail: int

class sc3nb.sc_objects.node.**Node**(*, *nodeid: Optional[int] = None*, *add_action: Optional[Union[*AddAction*, int]] = None*, *target: Optional[Union[*Node*, int]] = None*, *server: Optional[*sc3nb.sc_objects.server.SCServer*] = None*)

Bases: abc.ABC

Representation of a Node on SuperCollider.

Create a new Node

Parameters

nodeid
[int or None] This Nodes node id or None

add_action
[AddAction or corresponding int, optional] This Nodes AddAction when created in Server, by default None

target
[Node or int or None, optional] This Nodes AddActions target, by default None

server
[SCServer, optional] The Server for this Node, by default use the SC default server

Overview:

new	Create a new Node
_get_status_repr	
_set_node_attrs	Derive Node group from addaction and target
free	Free the node with /n_free.
run	Turn node on or off with /n_run.
set	Set a control value(s) of the node with n_set.
_update_control	
_update_controls	
fill	Fill ranges of control values with n_fill.
map	Map a node's control to read from a bus using /n_map or /n_mapa.
release	Set gate as specified.
query	Sends an n_query message to the server.
trace	Trace a node.
move	Move this node
register	Register to be watched.
unregister	Unregister to stop being watched.
on_free	Callback that is executed when this Synth is freed
wait	Wait until this Node is freed
_parse_info	
_handle_notification	
__eq__	Return self==value.
_get_nodeid	Get the corresponding node id

abstract new(**args*, *add_action: Optional[Union[*AddAction*, int]] = None*, *target: Optional[Union[*Node*, int]] = None*, *return_msg: bool = False*, ***kwargs*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Create a new Node

Parameters

add_action
[AddAction or int, optional] Where the Node should be added, by default AddAction.TO_HEAD (0)

target
[Node or int, optional] AddAction target, if None it will be the default group of the server

_get_status_repr() → str

_set_node_attrs(*target: Optional[Union[*Node*, int]] = None*, *add_action: Optional[Union[*AddAction*, int]] = None*) → None

Derive Node group from addaction and target

Parameters

target
[int or Node] Target nodeid or Target Node of this Node's AddAction

add_action
[AddAction] AddAction of this Node, default AddAction.TO_HEAD (0)

free(*return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Free the node with /n_free.

This will set is_running and is_playing to false. Even when the message is returned to mimic the behavior of the SuperCollider Node See https://doc.sccode.org/Classes/Node.html#-freeMsg

Returns

Node or OSCMessage
self for chaining or OSCMessage when return_msg=True

run(*on: bool = True*, *return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Turn node on or off with /n_run.

Parameters

on
[bool] True for on, False for off, by default True

Returns

Node or OSCMessage
self for chaining or OSCMessage when return_msg=True

set(*argument: Union[str, Dict, List]*, **values: Any*, *return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Set a control value(s) of the node with n_set.

Parameters

argument
[str | dict | list] if string: name of control argument if dict: dict with argument, value pairs if list: use list as message content

value
[any, optional] only used if argument is string, by default None

Examples

```
>>> synth.set("freq", 400)
>>> synth.set({"dur": 1, "freq": 400})
>>> synth.set(["dur", 1, "freq", 400])
```

_update_control(*control: str*, *value: Any*) → None

_update_controls(*controls: Optional[Dict[str, Any]] = None*) → None

fill(*control: Union[str, int]*, *num_controls: int*, *value: Any*, *return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Fill ranges of control values with n_fill.

Parameters

control
[int or string] control index or name

num_controls
[int] number of control values to fill

value
[float or int] value to set

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

OSCMessage
if return_msg else self

map(*control: Union[str, int]*, *bus:* sc3nb.sc_objects.bus.Bus, *return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Map a node's control to read from a bus using /n_map or /n_mapa.

Parameters

control
[int or string] control index or name

bus
[Bus] control/audio bus

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

OSCMessage
if return_msg else self

release(*release_time: Optional[float] = None*, *return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Set gate as specified.

https://doc.sccode.org/Classes/Node.html#-release

Parameters

release_time
[float, optional] amount of time in seconds during which the node will release. If set to a value <= 0, the synth will release immediately. If None using its Envs normal release stage(s)

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

OSCMessage
if return_msg else self

query() → Union[*SynthInfo*, *GroupInfo*]

Sends an n_query message to the server.

The answer is send to all clients who have registered via the /notify command. Content of answer:

node ID the node's parent group ID previous node ID, -1 if no previous node. next node ID, -1 if no next node. 1 if the node is a group, 0 if it is a synth

if the node is a group:
ID of the head node, -1 if there is no head node. ID of the tail node, -1 if there is no tail node.

Returns

SynthInfo or GroupInfo
n_info answer. See above for content description

`trace`(*return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Trace a node.

Print out values of the inputs and outputs for one control period. If node is a group then print the node IDs and names of each node.

Parameters

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

Node or OSCMessage
if return_msg else self

`move`(*add_action: AddAction*, *another_node: Node*, *return_msg: bool = False*) → Union[*Node*, *sc3nb.osc.osc_communication.OSCMessage*]

Move this node

Parameters

add_action
[AddAction [TO_HEAD, TO_TAIL, AFTER, BEFORE]] What add action should be done.

another_node
[Node] The node which is the target of the add action

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

Node or OSCMessage
if return_msg this will be the OSCMessage, else self

Raises

ValueError
If a wrong AddAction was provided

`abstract register`()

Register to be watched.

`abstract unregister`()

Unregister to stop being watched.

`on_free`(*func*)

Callback that is executed when this Synth is freed

`wait`(*timeout: Optional[float] = None*) → None

Wait until this Node is freed

Raises

TimeoutError
If timeout was provided and wait timed out.

_parse_info(*nodeid: int*, *group: int*, *prev_nodeid: int*, *next_nodeid: int*, **rest: Sequence[int]*) → Union[*SynthInfo*, *GroupInfo*]

_handle_notification(*kind: str*, *info*) → None

__eq__(*other*)

Return self==value.

static _get_nodeid(*value: Union[*Node*, int]*) → int

Get the corresponding node id

Parameters

value
[Node or int] If a Node is provided it will get its nodeid If a int is provided it will be returned

Returns

int
nodeid

Raises

ValueError
When neither Node or int was provided

class sc3nb.sc_objects.node.**Synth**(*name: Optional[str] = None*, *controls: Dict[str, Any] = None*, ***, *nodeid: Optional[int] = None*, *new: bool = True*, *add_action: Optional[Union[*AddAction*, int]] = None*, *target: Optional[Union[*Node*, int]] = None*, *server: Optional[*sc3nb.sc_objects.server.SCServer*] = None*)

Bases: *Node*

Representation of a Synth on SuperCollider.

Create a Python representation of a SuperCollider synth.

Parameters

name
[str, optional] name of the synth to be created, by default "default"

controls
[dict, optional] synth control arguments, by default None

nodeid
[int, optional] ID of the node in SuperCollider, by default sc3nb will create one. Can be set to an existing id to create a Python instance of a running Node.

new
[bool, optional] True if synth should be created on the server, by default True Should be False if creating an instance of a running Node.

add_action
[AddAction or int, optional] Where the Synth should be added, by default AddAction.TO_HEAD (0)

target
[Node or int, optional] AddAction target, if None it will be the default group of the server

server
[SCServer] sc3nb SCServer instance

Raises

ValueError
Raised when synth can't be found via SynthDescLib.global

Examples

```
>>> scn.Synth(sc, "s1", {"dur": 1, "freq": 400})
```

Overview:

_update_synth_state	
new	Creates the synth on the server with s_new.
get	Get a Synth argument
seti	Set part of an arrayed control.
__getattr__	
__setattr__	Implement setattr(self, name, value).
__repr__	Return repr(self).

_update_synth_state(*name: Optional[str]*, *controls: Optional[dict]*)

new(*controls: Optional[dict] = None*, *add_action: Optional[Union[*AddAction*, int]] = None*, *target: Optional[Union[*Node*, int]] = None*, *, *return_msg: bool = False*) → Union[*Synth*, *sc3nb.osc.osc_communication.OSCMessage*]

Creates the synth on the server with s_new.

Attention: Here you create an identical synth! Same nodeID etc. - This will fail if there is already this nodeID on the SuperCollider server!

get(*control: str*) → Any

Get a Synth argument

This will request the value from scsynth with /s_get(n).

Parameters

control
[str] name of the Synth control argument

abstract seti(**args*)

Set part of an arrayed control.

__getattr__(*name*)

__setattr__(*name*, *value*)

Implement setattr(self, name, value).

__repr__() → str

Return repr(self).

class sc3nb.sc_objects.node.**Group**(*, *nodeid: Optional[int] = None*, *new: bool = True*, *parallel: bool = False*, *add_action:* AddAction *= AddAction.TO_HEAD*, *target: Optional[Union[*Node*, int]] = None*, *server: Optional[*sc3nb.sc_objects.server.SCServer*] = None*)

Bases: *Node*

Representation of a Group on SuperCollider.

Create a Python representation of a SuperCollider group.

Parameters

nodeid
[int, optional] ID of the node in SuperCollider, by default sc3nb will create one. Can be set to an existing id to create a Python instance of a running Node.

new
[bool, optional] True if synth should be created on the server, by default True Should be False if creating an instance of a running Node.

parallel
[bool, optional] If True create a parallel group, by default False

add_action
[AddAction or int, optional] Where the Group should be added, by default AddAction.TO_HEAD (0)

target
[Node or int, optional] AddAction target, if None it will be the default group of the server

server
[SCServer, optional] Server instance where this Group is located, by default use the SC default server

Overview:

_update_group_state	
new	Creates the synth on the server with g_new / p_new.
move_node_to_head	Move node to this groups head with g_head.
move_node_to_tail	Move node to this groups tail with g_tail.
free_all	Frees all nodes in the group with g_freeAll.
deep_free	Free all synths in this group and its sub-groups with g_deepFree.
dump_tree	Posts a representation of this group's node subtree with g_dumpTree.
query_tree	Send a g_queryTree message for this group.
*_repr_pretty_*	
__repr__	Return repr(self).

_update_group_state(*children: Optional[Sequence[*Node*]] = None*) → None

new(*add_action=AddAction.TO_HEAD*, *target=None*, ***, *parallel=None*, *return_msg=False*) → Union[*Group*, *sc3nb.osc.osc_communication.OSCMessage*]

Creates the synth on the server with g_new / p_new.

Attention: Here you create an identical group! Same nodeID etc. - This will fail if there is already this nodeID on the SuperCollider server!

Parameters

add_action
[AddAction or int, optional] where the group should be added, by default AddAction.TO_HEAD (0)

target
[Node or int, optional] add action target, by default 1

parallel
[bool, optional] If True use p_new, by default False

return_msg
[bool, optional] If ture return the OSCMessage instead of sending it, by default False

Returns

Group
self

`move_node_to_head`(*node*, *return_msg=False*)

Move node to this groups head with g_head.

Parameters

node
[Node] node to move

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

Group
self

`move_node_to_tail`(*node*, *return_msg=False*)

Move node to this groups tail with g_tail.

Parameters

node
[Node] node to move

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

Group
self

`free_all`(*return_msg=False*)

Frees all nodes in the group with g_freeAll.

Parameters

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

OSCMessage
if return_msg else self

deep_free(*return_msg=False*)

Free all synths in this group and its sub-groups with g_deepFree.

Sub-groups are not freed.

Parameters

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

OSCMessage
if return_msg else self

dump_tree(*post_controls=True*, *return_msg=False*)

Posts a representation of this group's node subtree with g_dumpTree.

Parameters

post_controls
[bool, optional] True for control values, by default False

return_msg
[bool, optional] If True return msg else send it directly, by default False

Returns

OSCMessage
if return_msg else self

query_tree(*include_controls=False*) → *Group*

Send a g_queryTree message for this group.

See https://doc.sccode.org/Reference/Server-Command-Reference.html#/g_queryTree for details.

Parameters

include_controls
[bool, optional] True for control values, by default False

Returns

tuple
/g_queryTree.reply

_repr_pretty_(*printer*, *cylce*)

__repr__() → str

Return repr(self).

class sc3nb.sc_objects.node.**NodeTree**(*info: Sequence[Any]*, *root_nodeid: int*, *controls_included: bool*, *start: int = 0*, *server: Optional[*sc3nb.sc_objects.server.SCServer*] = None*)

Node Tree is a class for parsing /g_queryTree.reply

Overview:

parse_nodes	Parse Nodes from reply of the /g_queryTree cmd of scsynth.
*_repr_pretty_*	

static parse_nodes(*info: Sequence[Any]*, *controls_included: bool = True*, *start: int = 0*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*) → Tuple[int, *Node*]

Parse Nodes from reply of the /g_queryTree cmd of scsynth. This reads the /g_queryTree.reply and creates the corresponding Nodes in Python. See https://doc.sccode.org/Reference/Server-Command-Reference.html#/g_queryTree

Parameters

controls_included
[bool] If True the current control (arg) values for synths will be included

start
[int] starting position of the parsing, used for recursion, default 0

info
[Sequence[Any]] /g_queryTree.reply to be parsed.

Returns

Tuple[int, Node]
postion where the parsing ended, resulting Node

_repr_pretty_(*printer*, *cylce*)

sc3nb.sc_objects.recorder

Module for recording

Module Contents

Class List

RecorderState	Different States
Recorder	Allows to record audio easily.

Content

class sc3nb.sc_objects.recorder.**RecorderState**

Bases: enum.Enum

Different States

UNPREPARED = 'UNPREPARED'

PREPARED = 'PREPARED'

RECORDING = 'RECORDING'

PAUSED = 'PAUSED'

class sc3nb.sc_objects.recorder.**Recorder**(*path: str = 'record.wav'*, *nr_channels: int = 2*, *rec_header: str = 'wav'*, *rec_format: str = 'int16'*, *bufsize: int = 65536*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

Allows to record audio easily.

Create and prepare a recorder.

Parameters

path
[str, optional] path of recording file, by default "record.wav"

nr_channels
[int, optional] Number of channels, by default 2

rec_header
[str, optional] File format, by default "wav"

rec_format
[str, optional] Recording resolution, by default "int16"

bufsize
[int, optional] size of buffer, by default 65536

server
[SCServer, optional] server used for recording, by default use the SC default server

Overview:

prepare	Pepare the recorder.
start	Start the recording.
pause	Pause the recording.
resume	Resume the recording
stop	Stop the recording.
__repr__	Return repr(self).
__del__	

prepare(*path: str = 'record.wav', nr_channels: int = 2, rec_header: str = 'wav', rec_format: str = 'int16', bufsize: int = 65536*)

Pepare the recorder.

Parameters

path
[str, optional] path of recording file, by default "record.wav"

nr_channels
[int, optional] Number of channels, by default 2

rec_header
[str, optional] File format, by default "wav"

rec_format
[str, optional] Recording resolution, by default "int16"

bufsize
[int, optional] size of buffer, by default 65536

Raises

RuntimeError
When Recorder does not needs to be prepared.

start(*timetag: float = 0*, *duration: Optional[float] = None*, *node: Union[sc3nb.sc_objects.node.Node, int] = 0*, *bus: int = 0*)

Start the recording.

Parameters

timetag
[float, by default 0 (immediately)] Time (or time offset when <1e6) to start

duration
[float, optional] Length of the recording, by default until stopped.

node
[Union[Node, int], optional] Node that should be recorded, by default 0

bus
[int, by default 0] Bus that should be recorded

Raises

RuntimeError
When trying to start a recording unprepared.

pause(*timetag: float = 0*)

Pause the recording.

Parameters

timetag
[float, by default 0 (immediately)] Time (or time offset when <1e6) to pause

Raises

RuntimeError
When trying to pause if not recording.

resume(*timetag: float = 0*)

Resume the recording

Parameters

timetag
[float, by default 0 (immediately)] Time (or time offset when <1e6) to resume

Raises

RuntimeError
When trying to resume if not paused.

stop(*timetag: float = 0*)

Stop the recording.

Parameters

timetag
[float, by default 0 (immediately)] Time (or time offset when <1e6) to stop

Raises

RuntimeError
When trying to stop if not started.

__repr__() → str

Return repr(self).

__del__()

sc3nb.sc_objects.score

Module for creating SuperCollider OSC files that can be used for non-realtime synthesis

SuperCollider Guide - Non-Realtime Synthesis

Module Contents

Class List

Score

Content

sc3nb.sc_objects.score.**NRT_SERVER_OPTIONS**

class sc3nb.sc_objects.score.**Score**

Overview:

load_file	Load a OSC file into a dict.
write_file	Write this score as binary OSC file for NRT synthesis.
record_nrt	Write an OSC file from the messages and wri

classmethod load_file(*path: Union[str, bytes, os.PathLike]*) → Dict[float, List[*sc3nb.osc.osc_communication.OSCMessage*]]

Load a OSC file into a dict.

Parameters

path
[Union[str, bytes, os.PathLike]] Path of the OSC file.

Returns

Dict[float, List[OSCMessage]]
dict with time tag as keys and lists of OSCMessages as values.

classmethod write_file(*messages: Dict[float, List[*sc3nb.osc.osc_communication.OSCMessage*]]*, *path: Union[str, bytes, os.PathLike]*, *tempo: float = 1*)

Write this score as binary OSC file for NRT synthesis.

Parameters

messages
[Dict[float, List[OSCMessage]]] Dict with times as key and lists of OSC messages as values

path
[Union[str, bytes, os.PathLike]] output path for the binary OSC file

tempo
[float] Times will be multiplied by 1/tempo

classmethod record_nrt(*messages: Dict[float, List[sc3nb.osc.osc_communication.OSCMessage]]*, *osc_path: str*, *out_file: str*, *in_file: Optional[str] = None*, *sample_rate: int = 44100*, *header_format: str = 'AIFF'*, *sample_format: str = 'int16'*, *options: Optional[sc3nb.sc_objects.server.ServerOptions] = None*)

Write an OSC file from the messages and wri

Parameters

messages
[Dict[float, List[OSCMessage]]] Dict with times as key and lists of OSC messages as values.

osc_path
[str] Path of the binary OSC file.

out_file
[str] Path of the resulting sound file.

in_file
[Optional[str], optional] Path of input soundfile, by default None.

sample_rate
[int, optional] sample rate for synthesis, by default 44100.

header_format
[str, optional] header format of the output file, by default "AIFF".

sample_format
[str, optional] sample format of the output file, by default "int16".

options
[Optional[ServerOptions], optional] instance of server options to specify server options, by default None

Returns

subprocess.CompletedProcess
Completed scsynth non-realtime process.

sc3nb.sc_objects.server

Module for managing Server related stuff.

Module Contents

Class List

MasterControlReply	Reply addresses of the Master Control Commands.
MasterControlCommand	Master Control commands of scsynth.
ReplyAddress	Specific reply addresses.
ServerStatus	Information about the status of the Server program
ServerVersion	Information about the version of the Server program
ServerOptions	Options for the SuperCollider audio server
NodeWatcher	The NodeWatcher is used to handle Node Notifications.
Hook	A simple class for storing a function and arguments and allows later execution.
SCServer	SuperCollider audio server representaion.

Content

sc3nb.sc_objects.server.**_LOGGER**

class sc3nb.sc_objects.server.**MasterControlReply**

Bases: str, enum.Enum

Reply addresses of the Master Control Commands.

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

VERSION_REPLY = '/version.reply'

SYNCED = '/synced'

STATUS_REPLY = '/status.reply'

class sc3nb.sc_objects.server.**MasterControlCommand**

Bases: str, enum.Enum

Master Control commands of scsynth.

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

DUMP_OSC = '/dumpOSC'

STATUS = '/status'

VERSION = '/version'

CLEAR_SCHED = '/clearSched'

NOTIFY = '/notify'

QUIT = '/quit'

SYNC = '/sync'

class sc3nb.sc_objects.server.**ReplyAddress**

Bases: str, enum.Enum

Specific reply addresses.

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

WILDCARD_ADDR = '/*'

FAIL_ADDR = '/fail'

DONE_ADDR = '/done'

RETURN_ADDR = '/return'

sc3nb.sc_objects.server.**ASYNC_CMDS**

sc3nb.sc_objects.server.**CMD_PAIRS**

sc3nb.sc_objects.server.**LOCALHOST = '127.0.0.1'**

sc3nb.sc_objects.server.**SC3NB_SERVER_CLIENT_ID = 1**

sc3nb.sc_objects.server.**SC3NB_DEFAULT_PORT = 57130**

sc3nb.sc_objects.server.**SCSYNTH_DEFAULT_PORT = 57110**

sc3nb.sc_objects.server.**SC3_SERVER_NAME = 'scsynth'**

class sc3nb.sc_objects.server.**ServerStatus**

Bases: NamedTuple

Information about the status of the Server program

num_ugens: int

num_synths: int

num_groups: int

num_synthdefs: int

avg_cpu: float

peak_cpu: float

nominal_sr: float

actual_sr: float

class sc3nb.sc_objects.server.**ServerVersion**

Bases: NamedTuple

Information about the version of the Server program

name: str

major_version: int

minor_version: int

patch_version: str

git_branch: str

commit: str

class sc3nb.sc_objects.server.**ServerOptions**(*udp_port: int = SCSYNTH_DEFAULT_PORT*, *max_logins: int = 6*, *num_input_buses: int = 2*, *num_output_buses: int = 2*, *num_audio_buses: int = 1024*, *num_control_buses: int = 4096*, *num_sample_buffers: int = 1024*, *publish_rendezvous: bool = False*, *block_size: Optional[int] = None*, *hardware_buffer_size: Optional[int] = None*, *hardware_sample_size: Optional[int] = None*, *hardware_input_device: Optional[str] = None*, *hardware_output_device: Optional[str] = None*, *other_options: Optional[Sequence[str]] = None*)

Options for the SuperCollider audio server

This allows the encapsulation and handling of the command line server options.

Overview:

__repr__	Return repr(self).

__repr__()

Return repr(self).

class sc3nb.sc_objects.server.**NodeWatcher**(*server: SCServer*)

The NodeWatcher is used to handle Node Notifications.

Parameters

server
[SCServer] Server belonging to the notifications

Overview:

handle_notification	Handle a Notification

handle_notification(**args*)

Handle a Notification

class sc3nb.sc_objects.server.**Hook**(*fun: Callable*, **args: Any*, ***kwargs: Any*)

A simple class for storing a function and arguments and allows later execution.

Create a Hook

Parameters

fun
[Callable] Function to be executed

args
[Any, optional] Arguments given to function

kwargs
[Any, optional] Keyword arguments given to function

Overview:

execute	Execute the Hook

execute()

Execute the Hook

class sc3nb.sc_objects.server.**SCServer**(*options: Optional[*ServerOptions*] = None*)

Bases: *sc3nb.osc.osc_communication.OSCCommunication*

SuperCollider audio server representaion.

Parameters

options
[Optional[ServerOptions], optional] Options used to start the local server, by default None

Create an OSC communication server

Parameters

server_ip
[str] IP address to use for this server

server_port
[int] port to use for this server

default_receiver_ip
[str] IP address used for sending by default

default_receiver_port
[int] port used for sending by default

Overview:

boot	Start the Server process.
init	Initialize the server.
execute_init_hooks	Run all init hook functions.
connect_sclang	Connect sclang to the server
add_init_hook	Create and add a hook to be executed when the server is initialized
remove_init_hook	Remove a previously added init Hook
bundler	Generate a Bundler with added server latency.
blip	Make a blip sound
remote	Connect to remote Server
reboot	Reboot this server
ping	Ping the server.
quit	Quits and tries to kill the server.
sync	Sync the server with the /sync command.
send_synthdef	Send a SynthDef as bytes.
load_synthdef	Load SynthDef file at path.
load_synthdefs	Load all SynthDefs from directory.
notify	Notify the server about this client.
free_all	Free all node ids.
clear_schedule	Send /clearSched to the server.
send_default_groups	Send the default groups for all clients.
mute	Mute audio
unmute	Set volume back to volume prior to muting

continues on next page

Table 2 – continued from previous page

version	Server version information
status	Server status information
dump_osc	Enable dumping incoming OSC messages at the server process
dump_tree	Server process prints out current nodes
query_tree	Query all nodes at the server and return a NodeTree
_init_osc_communication	
_get_errors_for_address	
_log_repr	
_log_message	
_warn_fail	
__repr__	Return repr(self).

boot(*scsynth_path: Optional[str] = None*, *timeout: float = 5*, *console_logging: bool = True*, *with_blip: bool = True*, *kill_others: bool = True*, *allowed_parents: Sequence[str] = ALLOWED_PARENTS*)

Start the Server process.

Parameters

scsynth_path
[str, optional] Path of scscynth executable, by default None

timeout
[float, optional] Timeout for starting the executable, by default 5

console_logging
[bool, optional] If True write process output to console, by default True

with_blip
[bool, optional] make a sound when booted, by default True

kill_others
[bool] kill other SuperCollider server processes.

allowed_parents
[Sequence[str], optional] Names of parents that are allowed for other instances of scsynth processes that won't be killed, by default ALLOWED_PARENTS

Raises

ValueError
If UDP port specified in options is already used

ProcessTimeout
If the process fails to start.

init(*with_blip: bool = True*)

Initialize the server.

This adds allocators, loads SynthDefs, send default Groups etc.

Parameters

with_blip
[bool, optional] make a sound when initialized, by default True

`execute_init_hooks`() → None

Run all init hook functions.

This is automatically done when running free_all, init or connect_sclang.

Hooks can be added using add_init_hook

`connect_sclang`(*port: int*) → None

Connect sclang to the server

This will add the "sclang" receiver and execute the init hooks

Parameters

port
[int] Port of sclang (NetAddr.langPort)

`add_init_hook`(*fun: Callable*, **args: Any*, ***kwargs: Any*) → *Hook*

Create and add a hook to be executed when the server is initialized

Parameters

hook
[Callable[..., None]] Function to be executed

args
[Any, optional] Arguments given to function

kwargs
[Any, optional] Keyword arguments given to function

Returns

Hook
The created Hook

`remove_init_hook`(*hook:* Hook)

Remove a previously added init Hook

Parameters

hook
[Hook] the hook to be removed

`bundler`(*timetag=0*, *msg=None*, *msg_params=None*, *send_on_exit=True*)

Generate a Bundler with added server latency.

This allows the user to easly add messages/bundles and send it.

Parameters

timetag
[float] Time at which bundle content should be executed. This servers latency will be added upon this. If timetag <= 1e6 it is added to time.time().

msg_addr
[str] SuperCollider address.

msg_params
[list, optional]

List of parameters to add to message.
(Default value = None)

Returns

Bundler
bundler for OSC bundling.

`blip`() → None

Make a blip sound

`remote`(*address: str*, *port: int*, *with_blip: bool = True*) → None

Connect to remote Server

Parameters

address
[str] address of remote server

port
[int] port of remote server

with_blip
[bool, optional] make a sound when initialized, by default True

`reboot`() → None

Reboot this server

Raises

RuntimeError
If this server is remote and can't be restarted.

`abstract ping`()

Ping the server.

`quit`() → None

Quits and tries to kill the server.

`sync`(*timeout=5*) → bool

Sync the server with the /sync command.

Parameters

timeout
[int, optional]

Time in seconds that will be waited for sync.
(Default value = 5)

Returns

bool
True if sync worked.

`send_synthdef`(*synthdef_bytes: bytes*)

Send a SynthDef as bytes.

Parameters

synthdef_bytes
[bytes] SynthDef bytes

wait
[bool] If True wait for server reply.

`load_synthdef`(*synthdef_path: str*)

Load SynthDef file at path.

Parameters

synthdef_path
[str] Path with the SynthDefs

bundle
[bool] Wether the OSC Messages can be bundle or not. If True sc3nb will not wait for the server response, by default False

`load_synthdefs`(*synthdef_dir: Optional[str] = None*, *completion_msg: Optional[bytes] = None*) → None

Load all SynthDefs from directory.

Parameters

synthdef_dir
[str, optional] directory with SynthDefs, by default sc3nb default SynthDefs

completion_msg
[bytes, optional] Message to be executed by the server when loaded, by default None

`notify`(*receive_notifications: bool = True*, *client_id: Optional[int] = None*, *timeout: float = 1*) → None

Notify the server about this client.

This provides the client id and max logins info needed for default groups.

Parameters

receive_notifications
[bool, optional] Flag for receiving node notification from server, by default True

client_id
[int, optional] Propose a client id, by default None

timeout
[float, optional] Timeout for server reply, by default 1.0

Raises

RuntimeError
If server has too many users.

OSCCommunicationError
If OSC communication fails.

`free_all`(*root: bool = True*) → None

Free all node ids.

Parameters

root
[bool, optional] If False free only the default group of this client, by default True

`clear_schedule`()

Send /clearSched to the server.

This clears all scheduled bundles and removes all bundles from the scheduling queue.

send_default_groups() → None

Send the default groups for all clients.

mute() → None

Mute audio

unmute() → None

Set volume back to volume prior to muting

version() → *ServerVersion*

Server version information

status() → *ServerStatus*

Server status information

dump_osc(*level: int = 1*) → None

Enable dumping incoming OSC messages at the server process

Parameters

level
[int, optional] Verbosity code, by default 1 0 turn dumping OFF. 1 print the parsed contents of the message. 2 print the contents in hexadecimal. 3 print both the parsed and hexadecimal representations.

dump_tree(*controls: bool = True*, *return_tree=False*) → Optional[str]

Server process prints out current nodes

Parameters

controls
[bool, optional] If True include control values, by default True

return_tree
[bool, optional] If True return output as string, by default False

Returns

str
If return_tree this is the node tree string.

query_tree(*include_controls: bool = True*) → *sc3nb.sc_objects.node.Group*

Query all nodes at the server and return a NodeTree

Parameters

include_controls
[bool, optional] If True include control values, by default True

Returns

NodeTree
object containing all the nodes.

_init_osc_communication()

_get_errors_for_address(*address: str*)

_log_repr()

_log_message(*sender*, **params*)

_warn_fail(*sender*, **params*)

__repr__() → str
Return repr(self).

sc3nb.sc_objects.synthdef

Module to for using SuperCollider SynthDefs and Synths in Python

Module Contents

Class List

SynthDefinitionCommand	OSC Commands for Synth Definitions
SynthDef	Wrapper for SuperCollider SynthDef

Content

class sc3nb.sc_objects.synthdef.**SynthDefinitionCommand**

Bases: str, enum.Enum

OSC Commands for Synth Definitions

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

RECV = '/d_recv'

LOAD = '/d_load'

LOAD_DIR = '/d_loadDir'

FREE = '/d_free'

class sc3nb.sc_objects.synthdef.**SynthDef**(*name: str*, *definition: str*, *sc: Optional[*sc3nb.sc.SC*] = None*)

Wrapper for SuperCollider SynthDef

Create a dynamic synth definition in sc.

Parameters

name
[string] default name of the synthdef creation. The naming convention will be name+int, where int is the amount of already created synths of this definition

definition
[string] Pass the default synthdef definition here. Flexible content should be in double brackets (”...{{flexibleContent}}...”). This flexible content, you can dynamic replace with set_context()

sc
[SC object] SC instance where the synthdef should be created, by default use the default SC instance

synth_descs

synth_defs

Overview:

get_description	Get Synth description
send	Send a SynthDef as bytes.
load	Load SynthDef file at path.
load_dir	Load all SynthDefs from directory.
reset	Reset the current synthdef configuration to the self.definition value.
set_context	Set context in SynthDef.
set_contexts	Set multiple values at onces when you give a dictionary.
unset_remaining	This method will remove all existing placeholders in the current def.
add	This method will add the current_def to SuperCollider.s
free	Free this SynthDef from the server.
__repr__	Return repr(self).

classmethod get_description(*name: str*, *lang: Optional[sc3nb.sclang.SCLang] = None*) → Optional[Dict[str, *sc3nb.sclang.SynthArgument*]]

Get Synth description

Parameters

name
[str] name of SynthDef

Returns

Dict
dict with SynthArguments

classmethod send(*synthdef_bytes: bytes*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

Send a SynthDef as bytes.

Parameters

synthdef_bytes
[bytes] SynthDef bytes

wait
[bool] If True wait for server reply.

server
[SCServer, optional] Server instance that gets the SynthDefs, by default use the SC default server

classmethod load(*synthdef_path: str*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

Load SynthDef file at path.

Parameters

synthdef_path
[str] Path with the SynthDefs

server
[SCServer, optional] Server that gets the SynthDefs, by default use the SC default server

classmethod load_dir(*synthdef_dir: Optional[str] = None*, *completion_msg: Optional[bytes] = None*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

Load all SynthDefs from directory.

Parameters

synthdef_dir
[str, optional] directory with SynthDefs, by default sc3nb default SynthDefs

completion_msg
[bytes, optional] Message to be executed by the server when loaded, by default None

server
[SCServer, optional] Server that gets the SynthDefs, by default use the SC default server

reset() → *SynthDef*

Reset the current synthdef configuration to the self.definition value.

After this you can restart your configuration with the same root definition

Returns

object of type SynthDef
the SynthDef object

set_context(*searchpattern: str*, *value*) → *SynthDef*

Set context in SynthDef.

This method will replace a given key (format: "...{{key}}...") in the synthdef definition with the given value.

Parameters

searchpattern
[string] search pattern in the current_def string

value
[string or something with can parsed to string] Replacement of search pattern

Returns

self
[object of type SynthDef] the SynthDef object

set_contexts(*dictionary: Dict[str, Any]*) → *SynthDef*

Set multiple values at onces when you give a dictionary.

Because dictionaries are unsorted, keep in mind, that the order is sometimes ignored in this method.

Parameters

dictionary
[dict] {searchpattern: replacement}

Returns

self
[object of type SynthDef] the SynthDef object

unset_remaining() → *SynthDef*

This method will remove all existing placeholders in the current def.

You can use this at the end of definition to make sure, that your definition is clean. Hint: This method will not remove pyvars

Returns

self
[object of type SynthDef] the SynthDef object

add(*pyvars=None*, *name: Optional[str] = None*, *server: Optional[sc3nb.sc_objects.server.SCServer] = None*) → str

This method will add the current_def to SuperCollider.s

If a synth with the same definition was already in sc, this method will only return the name.

Parameters

pyvars
[dict] SC pyvars dict, to inject python variables

name
[str, optional] name which this SynthDef will get

server
[SCServer, optional] Server where this SynthDef will be send to, by default use the SC default server

Returns

str
Name of the SynthDef

free() → *SynthDef*

Free this SynthDef from the server.

Returns

self
[object of type SynthDef] the SynthDef object

__repr__()

Return repr(self).

sc3nb.sc_objects.volume

Server Volume controls.

Module Contents

Class List

Volume	Server volume controls

Content

sc3nb.sc_objects.volume._LOGGER

class sc3nb.sc_objects.volume.Volume(*server:* sc3nb.sc_objects.server.SCServer, *min_:* *int* = *-90*, *max_:* *int* = *6*)

Server volume controls

Overview:

mute	Mute audio
unmute	Unmute audio
update_volume_synth	Update volume Synth
send__volume_synthdef	Send Volume SynthDef

mute() → None

Mute audio

unmute() → None

Unmute audio

update_volume_synth() → None

Update volume Synth

send__volume_synthdef()

Send Volume SynthDef

12.3 Submodules

12.3.1 sc3nb.magics

This module adds the Jupyter specialties such as Magics and Keyboard Shortcuts

12.3.1.1 Module Contents

Function List

load_ipython_extension	Function that is called when Jupyter loads this as extension (%load_ext sc3nb)
add_shortcut	Add the server 'free all' shortcut.

Class List

SC3Magics	IPython magics for SC class

Content

sc3nb.magics.**load_ipython_extension**(*ipython*) → None

Function that is called when Jupyter loads this as extension (%load_ext sc3nb)

Parameters

ipython
[IPython] IPython object

sc3nb.magics.**add_shortcut**(*ipython*, *shortcut: str = None*) → None

Add the server 'free all' shortcut.

Parameters

ipython
[IPython] IPython object

shortcut
[str, optional] shortcut for 'free all', by default it is "cmd-." or "Ctrl-."

class sc3nb.magics.**SC3Magics**(*shell=None*, ***kwargs*)

Bases: IPython.core.magic.Magics

IPython magics for SC class

Create a configurable given a config config.

Parameters

config
[Config] If this is empty, default values are used. If config is a Config instance, it will be used to configure the instance.

parent
[Configurable instance, optional] The parent Configurable instance of this object.

Notes

Subclasses of Configurable must call the __init__() method of Configurable *before* doing anything else and using super():

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config=config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

Overview:

sc	Execute SuperCollider code via magic
scv	Execute SuperCollider code with verbose output
scs	Execute SuperCollider code silently (verbose==False)
scg	Execute SuperCollider code returning output
scgv	Execute SuperCollider code returning output
scgs	Execute SuperCollider code returning output
_parse_pyvars	Parses SuperCollider code for python variables and their values

sc(*line=''*, *cell=None*)

Execute SuperCollider code via magic

Parameters

line
[str, optional] Line of SuperCollider code, by default ‘’

cell
[str, optional] Cell of SuperCollider code , by default None

Returns

Unknown
cmd result

scv(*line=''*, *cell=None*)

Execute SuperCollider code with verbose output

Parameters

line
[str, optional] Line of SuperCollider code, by default ‘’

cell
[str, optional] Cell of SuperCollider code , by default None

Returns

Unknown
cmd result

scs(*line=''*, *cell=None*)

Execute SuperCollider code silently (verbose==False)

Parameters

line
[str, optional] Line of SuperCollider code, by default ‘’

cell
[str, optional] Cell of SuperCollider code , by default None

Returns

Unknown
cmd result

scg(*line=''*, *cell=None*)

Execute SuperCollider code returning output

Parameters

line
[str, optional] Line of SuperCollider code, by default ‘’

cell
[str, optional] Cell of SuperCollider code , by default None

Returns

Unknown
cmd result Output from SuperCollider code, not all SC types supported, see pythonosc.osc_message.Message for list of supported types

scgv(*line=''*, *cell=None*)

Execute SuperCollider code returning output

Parameters

line
[str, optional] Line of SuperCollider code, by default ‘’

cell
[str, optional] Cell of SuperCollider code , by default None

Returns

Unknown
cmd result Output from SuperCollider code, not all SC types supported, see pythonosc.osc_message.Message for list of supported types

scgs(*line=''*, *cell=None*)

Execute SuperCollider code returning output

Parameters

line
[str, optional] Line of SuperCollider code, by default ‘’

cell
[str, optional] Cell of SuperCollider code , by default None

Returns

Unknown
cmd result Output from SuperCollider code, not all SC types supported, see pythonosc.osc_message.Message for list of supported types

_parse_pyvars(*code: str*) → Dict[str, Any]

Parses SuperCollider code for python variables and their values

Parameters

code
[str] SuperCollider code snippet

Returns

Dict[str, Any]
Dict with variable names and their values.

Raises

NameError
If pyvar injection value can’t be found.

12.3.2 sc3nb.process_handling

Module for process handling.

12.3.2.1 Module Contents

Function List

find_executable	Looks for executable in os $PATH or specified path
kill_processes	Kill processes with the same path for the executable.

Class List

Process	Class for starting a executable and communication with it.

Content

sc3nb.process_handling.**_LOGGER**

sc3nb.process_handling.**ANSI_ESCAPE**

sc3nb.process_handling.**ALLOWED_PARENTS = ('scide', 'python', 'tox')**

sc3nb.process_handling.**find_executable**(*executable: str*, *search_path: str = None*, *add_to_path: bool = False*)

Looks for executable in os $PATH or specified path

Parameters

executable
[str] Executable to be found

search_path
[str, optional] Path at which to look for, by default None

add_to_path
[bool, optional] Wether to add the provided path to os $PATH or not, by default False

Returns

str
Full path to executable

Raises

FileNotFoundError
Raised if executable cannot be found

sc3nb.process_handling.**kill_processes**(*exec_path*, *allowed_parents: Optional[tuple] = None*)

Kill processes with the same path for the executable.

If allowed_parent is provided it will be searched in the names of the parent processes of the process with the executable path before terminating. If it is found the process won't be killed.

Parameters

exec_path
[str] path of the executable to kill

allowed_parent
[str, optional] parents name of processes to keep, by default None

exception sc3nb.process_handling.**ProcessTimeout**(*executable*, *output*, *timeout*, *expected*)

Bases: Exception

Process Timeout Exception

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

class sc3nb.process_handling.**Process**(*executable: str*, *programm_args: Optional[Sequence[str]] = None*, *executable_path: str = None*, *console_logging: bool = True*, *kill_others: bool = True*, *allowed_parents: Sequence[str] = None*)

Class for starting a executable and communication with it.

Parameters

executable
[str] Name of executable to start

programm_args
[Optional[Sequence[str]], optional] Arguments to program start with Popen, by default None

executable_path
[str, optional] Path with executalbe, by default system PATH

console_logging
[bool, optional] Flag for controlling console logging, by default True

kill_others
[bool, optional] Flag for controlling killing of other executables with the same name. This is useful when processes where left over, by default True

allowed_parents
[Sequence[str], optional] Sequence of parent names that won't be killed when kill_others is True, by default None

Overview:

_read_loop	
read	Reads current output from output queue until expect is found
empty	Empties output queue.
write	Send input to process
kill	Kill the process.
__del__	
__repr__	Return repr(self).

_read_loop()

read(*expect: Optional[str] = None*, *timeout: float = 3*) → str

Reads current output from output queue until expect is found

Parameters

expect
[str, optional] str that we expect to find, by default None

timeout
[float, optional] timeout in seconds for waiting for output, by default 3

Returns

str
Output of process.

Raises

ProcessTimeout
If neither output nor expect is found

`empty()` → None

Empties output queue.

`write(input_str: str)` → None

Send input to process

Parameters

input_str
[str] Input to be send to process

Raises

RuntimeError
If writing to process fails

`kill()` → int

Kill the process.

Returns

int
return code of process

`__del__()`

`__repr__()` → str

Return repr(self).

12.3.3 sc3nb.sc

Contains classes enabling the easy communication with SuperCollider within jupyter notebooks

12.3.3.1 Module Contents

Function List

startup	Inits SuperCollider (scsynth, sclang) and registers ipython magics

Class List

SC	Create a SuperCollider Wrapper object.

Content

sc3nb.sc._LOGGER

sc3nb.sc.**startup**(*start_server: bool = True*, *scsynth_path: Optional[str] = None*, *start_sclang: bool = True*, *sclang_path: Optional[str] = None*, *magic: bool = True*, *scsynth_options: Optional[sc3nb.sc_objects.server.ServerOptions] = None*, *with_blip: bool = True*, *console_logging: bool = False*, *allowed_parents: Sequence[str] = ALLOWED_PARENTS*, *timeout: float = 10*) → *SC*

Inits SuperCollider (scsynth, sclang) and registers ipython magics

Parameters

start_server
[bool, optional] If True boot scsynth, by default True

scsynth_path
[Optional[str], optional] Path of scscynth executable, by default None

start_sclang
[bool, optional] If True start sclang, by default True

sclang_path
[Optional[str], optional] Path of sclang executable, by default None

magic
[bool, optional] If True register magics to ipython, by default True

scsynth_options
[Optional[ServerOptions], optional] Options for the server, by default None

with_blip
[bool, optional] make a sound when booted, by default True

console_logging
[bool, optional] If True write scsynth/sclang output to console, by default False

allowed_parents
[Sequence[str], optional] Names of parents that are allowed for other instances of sclang/scsynth processes, by default ALLOWED_PARENTS

timeout
[float, optional] timeout in seconds for starting the executable, by default 10

Returns

SC
SuperCollider Interface class.

class sc3nb.sc.SC(*, *start_server: bool = True*, *scsynth_path: Optional[str] = None*, *start_sclang: bool = True*, *sclang_path: Optional[str] = None*, *scsynth_options: Optional[sc3nb.sc_objects.server.ServerOptions] = None*, *with_blip: bool = True*, *console_logging: bool = True*, *allowed_parents: Sequence[str] = ALLOWED_PARENTS*, *timeout: float = 5*)

Create a SuperCollider Wrapper object.

Parameters

start_server
[bool, optional] If True boot scsynth, by default True.

scsynth_path
[Optional[str], optional] Path of scscynth executable, by default None.

start_sclang
[bool, optional] If True start sclang, by default True.

sclang_path
[Optional[str], optional] Path of sclang executable, by default None.

scsynth_options
[Optional[ServerOptions], optional] Options for the server, by default None.

with_blip
[bool, optional] Make a sound when booted, by default True.

console_logging
[bool, optional] If True write scsynth/sclang output to console, by default True.

allowed_parents
[Sequence[str], optional] Names of parents that are allowed for other instances of sclang/scsynth processes, by default ALLOWED_PARENTS.

timeout
[float, optional] timeout in seconds for starting the executables, by default 5

default: Optional[SC]

Default SC instance.

This will be used by all SuperCollider objects if no SC/server/lang is specified.

Overview:

get_default	Get the default SC instance
start_sclang	Start this SuperCollider language
start_server	Start this SuperCollider server
_try_to_connect	
__del__	
__repr__	Return repr(self).
exit	Closes SuperCollider and shuts down server

classmethod get_default() → *SC*

Get the default SC instance

Returns

SC
default SC instance

Raises

RuntimeError
If there is no default SC instance.

start_sclang(*sclang_path: Optional[str] = None*, *console_logging: bool = True*, *allowed_parents: Sequence[str] = ALLOWED_PARENTS*, *timeout: float = 5*)

Start this SuperCollider language

Parameters

sclang_path
[Optional[str], optional] Path of sclang executable, by default None

console_logging
[bool, optional] If True write scsynth/sclang output to console, by default True

allowed_parents
[Sequence[str], optional] Names of parents that are allowed for other instances of sclang/scsynth processes, by default ALLOWED_PARENTS

timeout
[float, optional] timeout in seconds for starting the executable, by default 5

start_server(*scsynth_options: Optional[sc3nb.sc_objects.server.ServerOptions] = None*, *scsynth_path: Optional[str] = None*, *console_logging: bool = True*, *with_blip: bool = True*, *allowed_parents: Sequence[str] = ALLOWED_PARENTS*, *timeout: float = 5*)

Start this SuperCollider server

Parameters

scsynth_options
[Optional[ServerOptions], optional] Options for the server, by default None

scsynth_path
[Optional[str], optional] Path of scscynth executable, by default None

console_logging
[bool, optional] If True write scsynth/sclang output to console, by default True

with_blip
[bool, optional] make a sound when booted, by default True

allowed_parents
[Sequence[str], optional] Names of parents that are allowed for other instances of sclang/scsynth processes, by default ALLOWED_PARENTS

timeout
[float, optional] timeout in seconds for starting the executable, by default 5

_try_to_connect()

__del__()

__repr__() → str
Return repr(self).

exit() → None
Closes SuperCollider and shuts down server

12.3.4 sc3nb.sclang

Module for handling a SuperCollider language (sclang) process.

12.3.4.1 Module Contents

Class List

SynthArgument	Synth argument, rate and default value
SCLang	Class to control the SuperCollider Language Interpreter (sclang).

Content

sc3nb.sclang.**_LOGGER**

sc3nb.sclang.**SCLANG_DEFAULT_PORT = 57120**

sc3nb.sclang.**SC3NB_SCLANG_CLIENT_ID = 0**

class sc3nb.sclang.**SynthArgument**
Bases: NamedTuple

Synth argument, rate and default value

name: str

rate: str

default: Any

exception sc3nb.sclang.**SCLangError**(*message*, *sclang_output=None*)
Bases: Exception

Exception for Errors related to SuperColliders sclang.

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

class sc3nb.sclang.**SCLang**
Class to control the SuperCollider Language Interpreter (sclang).

Creates a python representation of sclang.

Raises

NotImplementedError
When an unsupported OS was found.

Overview:

start	Start and initilize the sclang process.
init	Initialize sclang for sc3nb usage.
load_synthdefs	Load SynthDef files from path.
kill	Kill this sclang instance.
__del__	

__repr__	Return repr(self).
cmd	Send code to sclang to execute it.
cmdv	cmd with verbose=True
cmds	cmd with verbose=False, i.e. silent
cmdg	cmd with get_result=True
read	Reads SuperCollider output from the process output queue.
empty	Empties sc output queue.
get_synth_description	Get a SynthDesc like description via sclang's global SynthDescLib.
connect_to_server	Connect this sclang instance to the SuperCollider server.

start(*sclang_path: Optional[str] = None*, *console_logging: bool = True*, *allowed_parents: Sequence[str] = ALLOWED_PARENTS*, *timeout: float = 10*) → None

Start and initilize the sclang process.

This will also kill sclang processes that does not have allowed parents.

Parameters

sclang_path
[Optional[str], optional] Path with the sclang executable, by default None

console_logging
[bool, optional] If True log sclang output to console, by default True

allowed_parents
[Sequence[str], optional] parents name of processes to keep, by default ALLOWED_PARENTS

timeout
[float, optional] timeout in seconds for starting the executable, by default 10

Raises

SCLangError
When starting or initilizing sclang failed.

init()

Initialize sclang for sc3nb usage.

This will register the /return callback in sclang and load the SynthDefs from sc3nb.

This is done automatically by running start.

load_synthdefs(*synthdefs_path: Optional[str] = None*) → None

Load SynthDef files from path.

Parameters

synthdefs_path
[str, optional] Path where the SynthDef files are located. If no path provided, load default sc3nb SynthDefs.

`kill()` → int

Kill this sclang instance.

Returns

int
returncode of the process.

`__del__()`

`__repr__()` → str

Return repr(self).

`cmd`(*code: str*, *pyvars: Optional[dict] = None*, *verbose: bool = True*, *discard_output: bool = True*, *get_result: bool = False*, *print_error: bool = True*, *get_output: bool = False*, *timeout: int = 1*) → Any

Send code to sclang to execute it.

This also allows to get the result of the code or the corresponding output.

Parameters

code
[str] SuperCollider code to execute.

pyvars
[dict, optional] Dictionary of name and value pairs of python variables that can be injected via ^name, by default None

verbose
[bool, optional] If True print output, by default True

discard_output
[bool, optional] If True clear output buffer before passing command, by default True

get_result
[bool, optional] If True receive and return the evaluation result from sclang, by default False

print_error
[bool, optional] If this and get_result is True and code execution fails the output from sclang will be printed.

get_output
[bool, optional] If True return output. Does not override get_result If verbose this will be True, by default False

timeout
[int, optional] Timeout in seconds for code execution return result, by default 1

Returns

Any

if get_result=True,
Result from SuperCollider code, not all SC types supported. When type is not understood this will return the datagram from the OSC packet.

if get_output or verbose
Output from SuperCollider code.

if get_output and get_result=True
(result, output)

else
None

Raises

RuntimeError
If get_result is True but no OSCCommunication instance is set.

SCLangError
When an error with sclang occurs.

cmdv(*code: str*, ***kwargs*) → Any

cmd with verbose=True

cmds(*code: str*, ***kwargs*) → Any

cmd with verbose=False, i.e. silent

cmdg(*code: str*, ***kwargs*) → Any

cmd with get_result=True

read(*expect: Optional[str] = None*, *timeout: float = 1*, *print_error: bool = True*) → str

Reads SuperCollider output from the process output queue.

Parameters

expect
[Optional[str], optional] Try to read this expected string, by default None

timeout
[float, optional] How long we try to read the expected string in seconds, by default 1

print_error
[bool, optional] If True this will print a message when timed out, by default True

Returns

str
output from sclang process.

Raises

timeout
If expected output string could not be read before timeout.

empty() → None

Empties sc output queue.

get_synth_description(*synth_def*)

Get a SynthDesc like description via sclang's global SynthDescLib.

Parameters

synth_def
[str] SynthDef name

Returns

dict
{argument_name: SynthArgument(rate, default)}

Raises

ValueError
When SynthDesc of synth_def can not be found.

connect_to_server(*server: Optional[sc3nb.sc_objects.server.SCServer] = None*)

Connect this sclang instance to the SuperCollider server.

This will set Server.default and s to the provided remote server.

Parameters

server
[SCServer, optional] SuperCollider server to connect. If None try to reconnect.

Raises

ValueError
If something different from an SCServer or None was provided

SCLangError
If sclang failed to register to the server.

12.3.5 sc3nb.timed_queue

Classes to run register functions at certain timepoints and run asynchronously

12.3.5.1 Module Contents

Class List

Event	Stores a timestamp, function and arguments for that function.
TimedQueue	Accumulates events as timestamps and functions.
TimedQueueSC	Timed queue with OSC communication.

Content

class sc3nb.timed_queue.**Event**(*timestamp: float, function: Callable[Ellipsis, None], args: Iterable[Any], spawn: bool = False*)

Stores a timestamp, function and arguments for that function. Long running functions can be wrapped inside an own thread

Parameters

timestamp
[float] Time event should be executed

function
[Callable[..., None]] Function to be executed

args
[Iterable[Any]] Arguments for function

spawn
[bool, optional] if True, create new thread for function, by default False

Overview:

execute	Executes function
__eq__	Return self==value.
__lt__	Return self<value.
__le__	Return self<=value.
__repr__	Return repr(self).

execute() → None

Executes function

__eq__(*other*)

Return self==value.

__lt__(*other*)

Return self<value.

__le__(*other*)

Return self<=value.

__repr__()

Return repr(self).

class sc3nb.timed_queue.**TimedQueue**(*relative_time: bool = False*, *thread_sleep_time: float = 0.001*, *drop_time_threshold: float = 0.5*)

Accumulates events as timestamps and functions.

Executes given functions according to the timestamps

Parameters

relative_time
[bool, optional] If True, use relative time, by default False

thread_sleep_time
[float, optional] Sleep time in seconds for worker thread, by default 0.001

drop_time_threshold
[float, optional] Threshold for execution time of events in seconds. If this is exceeded the event will be dropped, by default 0.5

Overview:

close	Closes event processing without waiting for pending events
join	Closes event processing after waiting for pending events
complete	Blocks until all pending events have completed
put	Adds event to queue
get	Get latest event from queue and remove event
peek	Look up latest event from queue
empty	Checks if queue is empty
pop	Removes latest event from queue
__worker	Worker function to process events
__repr__	Return repr(self).
elapse	Add time delta to the current queue time.

`close()` → None

Closes event processing without waiting for pending events

`join()` → None

Closes event processing after waiting for pending events

`complete()` → None

Blocks until all pending events have completed

`put`(*timestamp: float*, *function: Callable[Ellipsis, None]*, *args: Iterable[Any] = ()*, *spawn: bool = False*) → None

Adds event to queue

Parameters

timestamp
[float] Time (POSIX) when event should be executed

function
[Callable[..., None]] Function to be executed

args
[Iterable[Any], optional] Arguments to be passed to function, by default ()

spawn
[bool, optional] if True, create new sub-thread for function, by default False

Raises

TypeError
raised if function is not callable

`get()` → *Event*

Get latest event from queue and remove event

Returns

Event
Latest event

`peek()` → *Event*

Look up latest event from queue

Returns

Event
Latest event

`empty()` → bool

Checks if queue is empty

Returns

bool
True if queue if empty

`pop()` → None

Removes latest event from queue

`__worker`(*sleep_time: float*, *close_event: threading.Event*) → NoReturn

Worker function to process events

__repr__()

Return repr(self).

elapse(*time_delta: float*) → None

Add time delta to the current queue time.

Parameters

time_delta
[float] Additional time

class sc3nb.timed_queue.**TimedQueueSC**(*server: sc3nb.osc.osc_communication.OSCCommunication = None*, *relative_time: bool = False*, *thread_sleep_time: float = 0.001*)

Bases: *TimedQueue*

Timed queue with OSC communication.

Parameters

server
[OSCCommunication, optional] OSC server to handle the bundlers and messsages, by default None

relative_time
[bool, optional] If True, use relative time, by default False

thread_sleep_time
[float, optional] Sleep time in seconds for worker thread, by default 0.001

Overview:

put_bundler	Add a Bundler to queue
put_msg	Add a message to queue

put_bundler(*onset: float*, *bundler: sc3nb.osc.osc_communication.Bundler*) → None

Add a Bundler to queue

Parameters

onset
[float] Sending timetag of the Bundler

bundler
[Bundler] Bundler that will be sent

put_msg(*onset: float*, *msg: Union[sc3nb.osc.osc_communication.OSCMessage, str]*, *msg_params: Iterable[Any]*) → None

Add a message to queue

Parameters

onset
[float] Sending timetag of the message

msg
[Union[OSCMessage, str]] OSCMessage or OSC address

msg_params
[Iterable[Any]] If msg is str, this will be the parameters of the created OSCMessage

12.3.6 sc3nb.util

Module with utlilty functions - especially for handling code snippets

12.3.6.1 Module Contents

Function List

is_socket_used	

remove_comments	Removes all c-style comments from code.
parse_pyvars	Looks through call stack and finds values of variables.
replace_vars	Replaces python variables with SuperCollider literals in code.
convert_to_sc	Converts python objects to SuperCollider code literals.

Content

sc3nb.util.**is_socket_used**(*addr=('127.0.0.1', 57110)*)

sc3nb.util.**remove_comments**(*code: str*) → str

Removes all c-style comments from code.

This removes *//single-line* or */* multi-line */* comments.

Parameters

code
[str] Code where comments should be removed.

Returns

str
code string without comments

sc3nb.util.**parse_pyvars**(*code: str*, *frame_nr: int = 2*)

Looks through call stack and finds values of variables.

Parameters

code
[str] SuperCollider command to be parsed

frame_nr
[int, optional] on which frame to start, by default 2 (grandparent frame)

Returns

dict
{variable_name: variable_value}

Raises

NameError
If the variable value could not be found.

sc3nb.util.**replace_vars**(*code: str*, *pyvars: dict*) → str

Replaces python variables with SuperCollider literals in code.

This replaces the pyvars preceded with ^ in the code with a SC literal. The conversion is done with convert_to_sc.

Parameters

code
[str] SuperCollider Code with python injections.

pyvars
[dict] Dict with variable names and values.

Returns

str
Code with injected variables.

sc3nb.util.**convert_to_sc**(*obj: Any*) → str

Converts python objects to SuperCollider code literals.

This supports currently:

- numpy.ndarray -> SC Array representation
- complex type -> SC Complex
- **strings -> if starting with sc3: it will be used as SC code**
  if it starts with a (single escaped backward slash) it will be used as symbol else it will be inserted as string

For unsupported types the __repr__ will be used.

Parameters

obj
[Any] object that should be converted to a SuperCollider code literal.

Returns

str
SuperCollider Code literal

12.3.7 sc3nb.version

CHAPTER

THIRTEEN

HOW TO CONTRIBUTE

You can contribute by creating issues, making pull requests, improving the documentation or creating examples.

Please get in touch with us if you wish to contribute. We are happy to be involved in the discussion of new features and to receive pull requests and also looking forward to include notebooks using sc3nb in our examples.

We will honor contributors in our *Contributors list*. If you contribute with a pull request feel free to add yourself to the list.

13.1 Development Guidelines

To ensure standards are followed please install the `[dev]` extra. See *How to set up the testing and development environment*

We use:

- pre-commit to controll our standards for each commit.
- black and isort for formatting code.
- python type hints
- numpydoc docstring format. Also see Guide to NumPy Documentation

13.2 How to contributing an example notebook

Feel free to suggest new example notebooks using sc3nb for all possible applications like sonification, music making or sound design.

Please use

```
import sc3nb as scn
sc = scn.startup()
```

to import sc3nb and start the SC instance.

Also add `sc.exit()` to all notebooks at the end.

Please also try to make sure they work when using the doc generation script.

13.3 How to set up the testing and development environment

Additional dependencies for sc3nb can be installed via the following extras:

Install	Purpose
`[test]`	running tox for tests and other things
`[dev]`	using the pre-commit hooks and installing other useful tools for development
`[docs]`	building the docs directly, without tox (used by tox)
`[localtest]`	running pytest directly, without tox (used by tox)

Normally you should only need `[test]` and `[dev]` for contributing.

13.4 How to test

The following tests should all be successful.

- run tests

```
tox
```

- test build

```
pip install --upgrade build
python -m build
```

- test building the docs

  For building the documentation for the current branch use:

```
tox -e docs
```

  Controll the output in `build/docs/html/`

13.5 How to prepare a release

- run all *tests*
- update changelog

  should contain information about the commits between the versions

- create a new git tag for this version

  We use semantic versioning. Please always use 3 numbers like v1.0.0 for setuptools_scm

- clear build files

```
rm dist build
```

- build

```
pip install --upgrade build
python -m build
```

- upload testpypi

```
pip install --user --upgrade twine
python -m twine upload --repository testpypi dist/*
```

- check testpypi

```
pip install --index-url https://test.pypi.org/simple/ --extra-index-url https://
↪pypi.org/simple sc3nb
```

- upload pypi

```
python -m twine upload dist/*
```

- check pypi

```
pip install sc3nb
```

- build github-pages docs (after pushing the tag) See *How to build the docs*

13.6 How to build the docs

Create the gh-pages documentation with

```
tox -e docs -- --github --clean --commit
```

Controll the commit in the build folder (`build/docs/gh-pages/repo/`). Push the changes

CHAPTER

FOURTEEN

CONTRIBUTORS

Please look at *How to contribute* for information about how to contribute.

14.1 Authors

- Thomas Hermann
- Dennis Reinsch

14.2 Contributors

- Ferdinand Schlatt
- Fabian Kaupmann
- Micha Steffen Vosse

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

O

P

Q

R

S

T

U

V

W

Z