



European Organisation for Astronomical Research in the Southern Hemisphere

Programme: ELT

Project/WP: Instrumentation Framework

ELT ICS Framework - Sequencer - User Manual

Document Number: ESO-363358

Document Version: 4

Document Type: Manual (MAN)

Released on: 2024-12-11

Document Classification: Public

Owner:	Muñoz, Iván
Validated by PM:	Kornweibel, Nick
Validated by SE:	González Herrera, Juan Carlos
Validated by PE:	Biancat Marchet, Fabio
Approved by PGM:	Tamai, Roberto

Name



Release

This document corresponds to [seq](#)¹ v4.2.0.

Authors

Name	Affiliation
Muñoz, Iván	ESO/DOE/CSE
Moins, Christophe	ESO/DOE/CSE

Change Record from Previous Version

Affected Section(s)	Changes / Reason / Remarks
	See CRE ET-1517
All	All sections updated
2.8,3	New sections added

¹<https://gitlab.eso.org/ifw/seq>



Contents:

1	Introduction	5
1.1	Scope	5
1.2	Acronyms	5
1.3	Overview	5
1.4	Naming Conventions	6
2	Tutorial	7
2.1	Building Sequences	7
2.2	Very simple sequences	8
2.3	Executing Tasks in Parallel	10
2.4	Executing Tasks in a Loop	12
2.5	Embedding Sequencer Scripts	13
2.6	Observation Blocks and Templates	15
2.7	Accessing Template Variables	17
2.8	Inserting a basic dialog window	18
3	Interface to Observation Handling SW	20
3.1	Configuration	20
3.2	Simulation	20
3.3	Instrument package	21
4	Sequencer GUI	22
4.1	Overview	22
4.2	Using the Sequencer GUI	24
4.3	Configuration File	33
4.4	Sequencer server	34
5	A Deeper Look	35
5.1	Passing Arguments to Actions	35
5.2	Using partial functions	35
5.3	Runtime Flags	36
5.4	Summary building DAGs	37
5.5	Special Variables	38
5.6	Result Handling	38
5.7	Finding nodes	39
5.8	Nodes have context	40
5.9	Node Types	41
6	Good Practices	44
6.1	Writing Sequences	44
7	Sequencer Command Line Tools	46
7.1	The <i>seqtool</i> meta command	46
7.2	seqtool run	53



ELT ICS Framework - Sequencer - User Manual

Doc. Number:	ESO-363358
Doc. Version:	4
Released on:	2024-12-11
Page:	4 of 54

7.3 seqtool draw	53
----------------------------	----



1 Introduction

The Sequencer is a software component developed in the scope of the Instrument Control System Framework (ICS FW) as the generic tool for the execution of Observation Blocks (OB) and engineering scripts.

1.1 Scope

This document is the user manual for the ELT ICS Framework - Sequencer. The intended audience are ELT users, consortia developers or software quality assurance engineers.

1.2 Acronyms

DB	Database
CCS	Central Control System
ELT	Extremely Large Telescope
FCF	Function Control Framework
GUI	Graphical User Interface
ICS	Instrument Control System
OB	Observation blocks
OHS	Observation Handling Software

1.3 Overview

The sequencer shall support the execution of OBs and engineering scripts to automatize maintenance and operational activities such as the daily startup/shutdown of the telescope.

The sequencer can be seen, broadly, as comprised of three main components.

Sequencer Engine

Allows to load and execute *Sequencer scripts*

Sequencer API

Allows to define *Sequencer scripts*

Sequencer GUI

Displays and interact graphically with Sequences

This documents shows how to use the *Sequencer API* in order to build *Sequencer scripts*. A basic tutorial is given in *Tutorial*. A more detailed look is shown in *A Deeper Look*. The good practises and some advices are given in the *Good Practices*, this section is still being written.

The Sequencer package provides some command line tools, they are described in *Sequencer Command Line Tools*.



1.4 Naming Conventions

Observation Block

A high level view of telescope operations. An observation block is the smallest observational unit for a telescope. It is a rather complex entity, containing all information necessary to execute sequentially and without interruption a set of correlated exposures, involving a single target (i.e. a single telescope preset).

Sequencer Script

A sequencer script is a script that can be executed by the sequencer, i.e. a template or an engineering script. The sequencer script may have parameters whose values determine the exact execution behavior. The sequencer script defines the execution order and the sequencer steps.

Engineering Script

A high level procedure to carry out a specific engineering task. The engineering script *is a sequencer script*.

Sequencer Step

An entity containing executable code, defined within a sequencer script.

Template

An entity dealing with the setup and execution of an observation.



2 Tutorial

The purpose of the sequencer is to execute *Sequencer scripts*, either for engineering or science purposes.

The unit of execution is a single *step* which is just a normal python function or method with *no input parameters*.

Sequences are modeled as Directed Acyclic Graph (DAG). Each node in the graph can either be a simple *action* which just invokes a single sequencer *step* or a more complex node which contains a complete sequencer script.

This allows sequences to be grouped and nested freely. Ultimately they will execute *steps*.

The *Sequencer API* allows the creation of these graphs.

Note: all the examples below can be found in `seq/samples/src/seq/samples`.

2.1 Building Sequences

Sequencer scripts are modeled as DAGs, the Sequencer API allows to create nodes in the DAG. There are different node types that determine the way its children are scheduled for execution (e.g. Parallel, Sequential). The sequencer scripts can use the same or different node types depending on the purpose. select, mix and match the node type(s) that suits better your needs.

The *Sequencer Engine* expects to find either a module method or a specific *class name* in a module in order to construct a Sequencer script from it. The conventions are the following.

1. A module level `create_sequence()` function. See *Tutorial 1 (a.py)*.
2. A class named `Tpl` which must provide a static method `create_sequence`. See *Parallel tasks sample (b.py)*.

In either case, the return value is the root node of the graph being implemented.

The first convention is tried first, if no `create_sequence()` function is found, the second convention is attempted.

For very simple scripts, following the first convention is perfectly fine. For more complex scripts, e.g. a Sequencer script going to be parametrized to control multiple devices at the same time, the class approach is recommended.



Node Dependencies

Regarding the order of execution, the DAG edges allows to represent the nodes dependencies. Each node in the DAG depends, from its parent nodes. Meaning that a node will not be started until every node that precedes it has finished its own execution.

Node dependencies determines the execution order of the sequence steps. A node won't run until all its dependencies have finished. In the graph, a node dependency is seen as an incoming edge.

Chaining the basic node types `Sequence` and `Parallel` defines a dependency hierarchy.

`Sequence` nodes are executed one after the other, therefore each node depends on its predecessor. On the other hand, `Parallel` nodes indicates that all nodes it contains shall be executed together. By combining and chaining this two basic node types it is possible to express any dependency graph.

2.2 Very simple sequences

As mentioned before the simplest *step* is a python coroutine with *no input parameters* which is used to create an **Action** node.

Note: The no input parameter rule can be bypassed with strategies shows in *Passing Arguments to Actions*

In this case, we define a sequence that executes two steps, one after the other, namely *a()* and *b()*. Source codes is shown below.

Listing 1: Tutorial 1 (a.py)

```
#!/usr/bin/env python3
"""
Simple example (A)

Executes function a() and b()
"""
import logging
import asyncio
from seq.lib.nodes import Sequence
from seq.lib import getUserLogger

#LOGGER = logging.getLogger(__name__)
LOGGER = getUserLogger()

async def a():
    """Simply a"""
```

(continues on next page)



(continued from previous page)

```
await asyncio.sleep(1)
LOGGER.info("Ax")
print("AAAA")
return "A"

async def b():
    """b example
    """
    LOGGER.info("Bx")
    print("BBBB")
    return "B"

def create_sequence(*args, **kw):
    """Builds my example sequence"""
    myname = kw.pop('name', "Example A")
    return Sequence.create(a, b, name=myname, **kw)
```

Important: The Sequencer Engine is based on the asyncio library, therefore it is *biased* towards coroutines, but they are not mandatory as shown in *Loop example (loop1.py)*.

These are some simple rules to create sequences:

- A step (Action) is a python coroutine *with no input parameters*. See *Passing Arguments to Actions* to break this rule.
- In this case, the sequence is created using the *Sequence.create* (Sequence) constructor, which receives the *steps* to be executed (in the given order).
- The python module which contains the sequence must define a **create_sequence** function as shown in the example. It returns a Sequence node that holds the nodes it will execute.

Important: The *Sequence.create* constructor provides syntax sugar in order to support passing coroutines as the sequence's graph nodes. In such a case a node of type `seq.nodes.Action` is *automatically* created and inserted in the graph.

This *simple sequence 01* is graphically shown below. It can be imported from python as:

```
>>> import seq.samples.a
```

Notice that only two nodes were specified in the *create_sequence()* function. However the *simple sequence 01* figure shows four nodes. The sequencer engine adds a *start node* (black circle) and *end node* (double black circle) to every node container type, i.e. those nodes that have children: Parallel, Sequence and Loop.

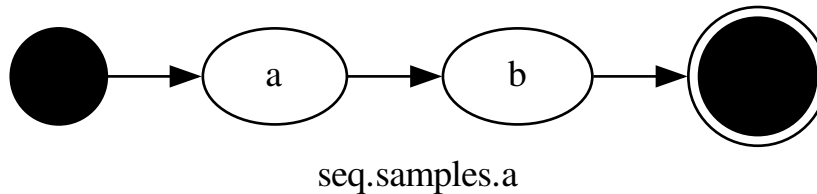


Fig. 1: simple sequence 01

Note: The *start* and *end* node, among other things, makes easy to chain nodes together by linking the end node of a container with the initial node of the next.

2.3 Executing Tasks in Parallel

One is not limited to create just linear sequences. Parallel activities (pseudo parallel) can be created using the `Parallel.create()` constructor. It receives the same parameters as the `Sequence` node constructor. When executed, the sequencer engine processes the `Parallel` nodes children in parallel.

Listing 2: Parallel tasks sample (b.py)

```
"""
Parallel nodes example.
"""
import asyncio
import random
import time
import logging
from seq.lib.nodes import Parallel, ActionInThread
Logger = logging.getLogger(__name__)

class Tpl:
    """A sample Sequence"""

    def a(self):
        """sleeps randomly"""
        t = random.randrange(5)
        time.sleep(t)
```

(continues on next page)



(continued from previous page)

```
Logger.info("... done A")

async def b(self):
    """sleeps randomly"""
    t = random.randrange(5)
    await asyncio.sleep(t)
    Logger.info("... done B")

@staticmethod
def create(*args, **kw):
    """Builds my sequence"""
    a = Tpl()
    p = Parallel.create( ActionInThread(a.a), a.b, **kw)
    return p
```

Which is represented graphically as follows.

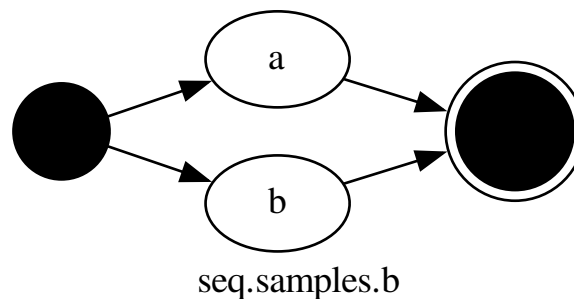


Fig. 2: Parallel Sequence

Points to notice:

1. In this case the Sequencer Engine discover a class named *Tpl* and calls its *create* method (@staticmethod as the convention mandates).
2. The example *Parallel Sequence* also shows that steps are not limited to coroutines. Just wrap it in *ActionInThread* node.
3. There is no problem mixing normal routines and asynchronous code. The sequencer will send the normal code to a separate thread and execute it there.

In order to avoid normal methods or functions to potentially block the *asyncio loop* (by holding the



CPU) they must be executed on their own *Thread*.

This is achieved with the `ActionInThread` node. In the example the `a()` method is wrapped in such node.

2.4 Executing Tasks in a Loop

The `Loop` node allows to repeat a set of steps while a condition is *True*.

Listing 3: Loop example (loop1.py)

```
"""
Implements a loop.
The condition checks Loop's index < 3.
"""

import asyncio
import logging
import random
from seq.lib.nodes import Loop

logger = logging.getLogger(__name__)

class Tpl: # Mandatory class name
    async def a(self):
        """sleeps up to 1 second"""
        t = random.random() # 0..1
        await asyncio.sleep(t)
        logger.info(".. done A: %d", Loop.index.get())

    async def b(self):
        """sleeps up to 1 second"""
        t = random.random() # 0..1
        await asyncio.sleep(t)
        logger.info(" .. done B: %d", Loop.index.get())

    async def c(self):
        pass

    async def check_condition(self):
        """
        The magic of contextvars in asyncio
        Loop.index is local to each asyncio task
        """
        logger.info("Loop index: %d", Loop.index.get())
```

(continues on next page)



(continued from previous page)

```
return Loop.index.get() < 3

@staticmethod
def create(**kw):
    t = Tpl()
    l = Loop.create(t.a, t.b, t.c,
                  condition=t.check_condition, **kw)

    return l
```

Which is represented graphically as follows.

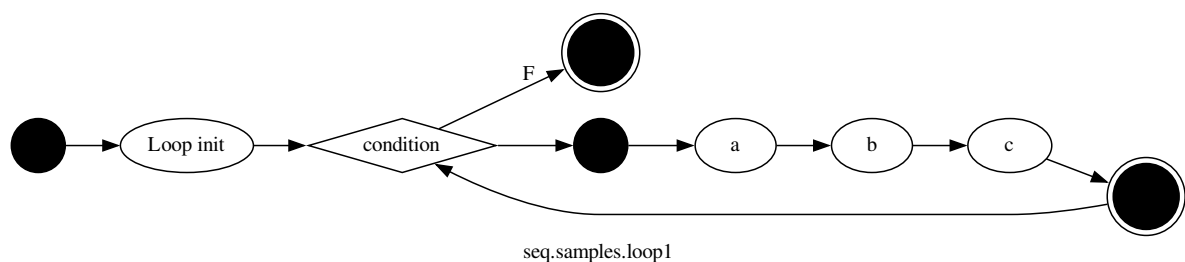


Fig. 3: Loop example

The code in *Loop example* shows the Loop's node constructor which takes a variable number of arguments comprising the Loop's body, i.e. the part that is repeated, *a()*, *b()* and *c()* in this case. The condition node is specified with the *condition* keyword.

The Loop's index is kept in the *context variable index*, meaning it can be accessed as *Loop.index.get()* as the *check_condition()* function shows.

In order to separate the index value of the different loops that might be occurring at the same time the Loop's index is implemented as an *asyncio context variable*. Therefore to get its value one has to call its *get()* method as the *check_condition()* function shows.

2.5 Embedding Sequencer Scripts

Sequences can be reused or embedded in order to produce more complex activities. The following example uses the sequences "a" and "b" to create a new sequence that executes them in *Parallel* and adds a step from a local class.

Important: Embedding a Sequence entails to import the module and instantiate its Sequencer script (either with *create_sequence()* or by *Tpl.create()*). The OB object can decide which one should be imported:



```
from seq.lib.nodes import seq_factory
from seq.samples.a

mynode = seq_factory(a)
```

Listing 4: Sequence embedding example

```
#!/usr/bin/env python3
"""
Simple example.

Uses nodes from template defined in module 'a'.
It also uses the 'a' template as a whole.
"""
from seq.lib.nodes import Parallel, seq_factory
from seq.lib.ob import OB

from seq.samples import a
from seq.samples import b

class Tpl:
    async def one(self):
        print("one")
        return 0

    async def two(self):
        print("two")
        return 99

    @staticmethod
    def create():
        aa = OB.create_sequence(a, name="A")
        bb = OB.create_sequence(b, name="B")

        #aa = seq_factory(a, name="A")
        #bb = seq_factory(b, name="B")
        s = Tpl()
        return Parallel.create(aa, bb, s.one)
```

Some points to note:

- Use `seq.lib.nodes.seq_factory()` to select the right method to instantiate a predefined sequencer script, so it can be reused.



2.6 Observation Blocks and Templates

Observation Blocks are defined through *JSON* files. A *JSON* file can store simple data structures and objects.

From the point of view of the Sequencer, an *Observation Block* is a sequence of templates that needs to be executed in the specified order. Therefore the sequencer is only concerned with the “templates” section of the *JSON* file. The *Sample Observation Block file* shows a simple *Observation Block*.

Listing 5: Sample Observation Block file

```
{
  "obId": 0,
  "itemType": "OB",
  "name": "obex 2",
  "executionTime": 0,
  "runId": "string",
  "instrument": "string",
  "ipVersion": "string",
  "obsDescription": {
    "name": "My humble II OB(A)",
    "userComments": "A",
    "instrumentComments": "AA"
  },
  "constraints": {
    "name": "string",
    "seeing": 0,
    "airmass": 0,
    "moonDistance": 0,
    "waterVapour": 0,
    "atm": "string",
    "fli": 0,
    "strehlRatio": 0,
    "skyTransparency": "string",
    "twilight": 0
  },
  "target": {
    "name": "string",
    "ra": "string",
    "dec": "string",
    "coordinateSystem": "ICRS",
    "comment": "string"
  },
  "templates": [
    {
```

(continues on next page)



(continued from previous page)

```
"templateName": "seq.samples.tpa",
"type": "string",
"parameters": [
  {
    "name": "par_b",
    "type": "integer",
    "value": 0
  },
  {
    "name": "par_c",
    "type": "number",
    "value": 77
  }
]
},
{
  "templateName": "seq.samples.tpa",
  "type": "string",
  "parameters": [
    {
      "name": "par_b",
      "type": "integer",
      "value": 0
    },
    {
      "name": "par_c",
      "type": "number",
      "value": 10
    }
  ]
}
],
"pi": {
  "firstName": "I",
  "lastName": "Condor",
  "email": "user@example.com"
}
}
```

In the current version, only the templates section is used. The Python modules implementing the desired actions are defined with the *templateName* keyword. Many templates can be specified this way. Moreover, many instances of the same template can be requested, The samples JSON file shows this case with the *seq.samples.tpa* Python module.



Note: *templateName* value must be a valid Python module expressed in *dot notation* similar to the import performed from the python prompt. Therefore the Python module must be available on the PYTHONPATH.

The *parameters* section describes the runtime parameters (name, type, value) to be handed over to the Template instance and they can be accessed from the Template's python code.

2.7 Accessing Template Variables

It is possible to access the value of the Template variables inside the Python code implementing a Template.

In order to access to template parameters like the ones defined in *Sample Observation Block file* one has to use the function *get_param()*. It knows the *current template* and request the parameters of interest. In the case of the example, the code to access *par_b* and *par_c* is as follows.

Listing 6: Access Template variables in Python

```
1  #!/usr/bin/env python3
2  """
3  Shows variables
4
5
6  """
7  import sys, inspect
8  import logging
9  import asyncio
10 from seq.lib.nodes import Sequence, Parallel, Template
11 from seq.lib.nodes import Action as _a
12 from seq.lib.nodes import ActionInThread as _ait
13 from seq.lib import logConfig
14 from seq.lib.nodes import get_param
15
16 import tkinter as tk
17 from tkinter import simpledialog
18
19
20 logger = logging.getLogger(__name__)
21
22 __seq_params__ = {"par_b": 11, "par_c": 22}
23
24 class Tpl:
25
```

(continues on next page)



(continued from previous page)

```
26  async def do_sum(self):
27      """Adds two integers"""
28      parb = get_param("par_b")
29      parc = get_param("par_c")
30
31      logger.debug("Node sum")
32      logger.info(
33          "xSUM {} + {} = {}".format(parb, parc, parb+parc))
34      await asyncio.sleep(1)
35
36  async def delay(self):
37      """Waits a sec"""
38      logger.debug("sleep a bit")
39      await asyncio.sleep(1)
40
41  @staticmethod
42  def create(*args, **kw):
43      """Adds two variables"""
44      a = Tpl()
45      return Sequence.create(
46          a.do_sum,
47          a.delay,
48          name="TPL_SUM Example", **kw)
49
50
51
```

2.8 Inserting a basic dialog window

A basic dialog window is displayed when a specified condition is not met. It gives the user the possibility to perform an additional step to overcome the unfulfilled condition and proceed or to stop the sequence execution. An instance of BasicDialog shall be inserted in a sequence as below with a condition and an extra step provided by the user:

Listing 7: Insertion of a basic dialog window in a sequence

```
#!/usr/bin/env python3
"""Usage of a dialog node node"""
import logging
import asyncio
from seq.lib.nodes import Sequence, BasicDialog
from seq.lib import logConfig
```

(continues on next page)



(continued from previous page)

```
LOGGER = logging.getLogger(__name__)

async def a():
    LOGGER.info("a")
    return "A"

async def b():
    LOGGER.info("b")

async def test_condition():
    await asyncio.sleep(0.5)
    return False

async def extra_step():
    await asyncio.sleep(3)
    print("What an extra step!")
    return True

def create_sequence(*args, **kw):
    """Builds my sequence"""
    logConfig(level=logging.INFO)
    c = BasicDialog(name="some_dialog", description="my first dialog", condition_label = "test_
↪condition", check_condition=test_condition, step_label="perform extra step", condition_extra_
↪step=extra_step)
    return Sequence.create(a,c,b, **kw)
```



3 Interface to Observation Handling SW

OTTO is an specification of the API used to fetch and execute OBs. It implements a REST interface to the Observation Handling SW.

OTTO services allows the Sequencer to:

- loading a visitor execution sequence (VES) into the UI.
- fetching an OB, aka “the next unit of execution”.
- reporting OB events Initiated, Started, Executed.

3.1 Configuration

In order to connect to OTTO server one needs its URL, username and password. This info is specified in the Sequencer GUI configuration file as:

```
otto:
  url: http://127.0.0.1:5000/
  insid: FORS2
  mode: VM
  user: pippo
  password: 123
```

The Sequencer GUI allows to modify the otto server params with a dedicated dialog box. See *Otto popup menu*. for details on OTTO and its GUI options and actions.

3.2 Simulation

TINO is the OTTO simulator. The recommended use is to start it on a terminal session. One can invoke it from *seqtool*. It starts a server with the OTTO interface from which OBs can be fetched. The OBs are served from a directory that contains them and it is a required argument of the *tino* subcommand. Tino's server port can also be specified from the command line.

TINO supports the following options:

```
$ seqtool tino --help
Usage: seqtool tino [OPTIONS] PATH

    Friend of OTTO

Options:
  --port INTEGER  Tino server port
  --help          Show this message and exit.
```



Usage

Upon starting TINO reports the address it is serving as:

```
$ seqtool tino ./OBs
Loading OB files:
* Serving Flask app "seq.otto.ottoSim" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Once TINO is running point the Sequencer GUI to the address TINO reported (normally *http://127.0.0.1:5000/*) see gui and OBs can be fetched from the server using the OTTO menu and 'Fetch OB' action.

Note: Regardless of the contents of the OB directory, TINO will only serve OBs that match the instrument configured.

3.3 Instrument package

An instrument package (IP) bundles instrument specific information. It will contain a set of template signatures, a set of observing constraints, a check list & questionnaire, external verification modules, execution time reporting modules and any other instrument-specific information. It is possible to fetch information from the template signatures IP and to integrate it in the ob for validation.

The CFGPATH shall point to a folder containing a directory named IP. IP will contain a directory named according to the instrument defined in the configuration. In the instrument directory, there will be a directory named ip that will contain two folders, app and templates. The app folder shall contain a file name library.cfg where a map is defined that associates a python module to each template. The signature of each of this template will be available in the templates folder.



4 Sequencer GUI

The sequencer GUI allows to load and execute Python sequences and OBs (JSON format as defined by OTTO).

Note: Do not rely on standard output to debug/check sequencer scripts.

Warning: As the server is responsible for executing sequencer scripts, the standard output is processed by the server.

4.1 Overview

The sequencer GUI requires its own special purpose *server*.

The server spawns and controls a *seqtool shell* process. The client(s) (*seqtool gui* in this case) talks to the server which relays the commands to the underlying *shell* process. This is *one way communication* from the client to the server.

However, the *seqtool shell* publishes the state of the Sequencer script execution, the modules loaded, state updates, logs and more using ELT's *Pub/Sub* mechanism and DDS. Allowing interested parties, namely *seqtool gui* or many instances of the *seqtool gui* to display the state Sequencer Graph. Figure *GUI interaction with server*.

The communication between the GUI and the server is via simple string commands over a TCP socket. The communication between the *server* and the *shell* is, again, simple string commands over a UNIX pipe (the server consumes and interprets shell's standard output) . Same for *shell* sending commands to the *kernel*.

It is inside the *seqtool kernel* process that the actual execution of the Sequencer scripts takes place, i.e. the kernel provides the *Sequencer engine*. It walks and executes the DAG constructed by the user using the *Sequencer API*.

The Sequencer GUI requires the *sequencer server* to be up and running in order to work. Upon starting, the GUI allocates and spawns its own server. On the other hand there is an option to startup the GUI and connect it to a running server.

During normal use, one would start the GUI (and all supporting processes) as:

```
$ seqtool gui
```

In such a case, a sequencer server will be spawned to which the gui will connect.

To connect to a running server, e.g. running at port 8000:

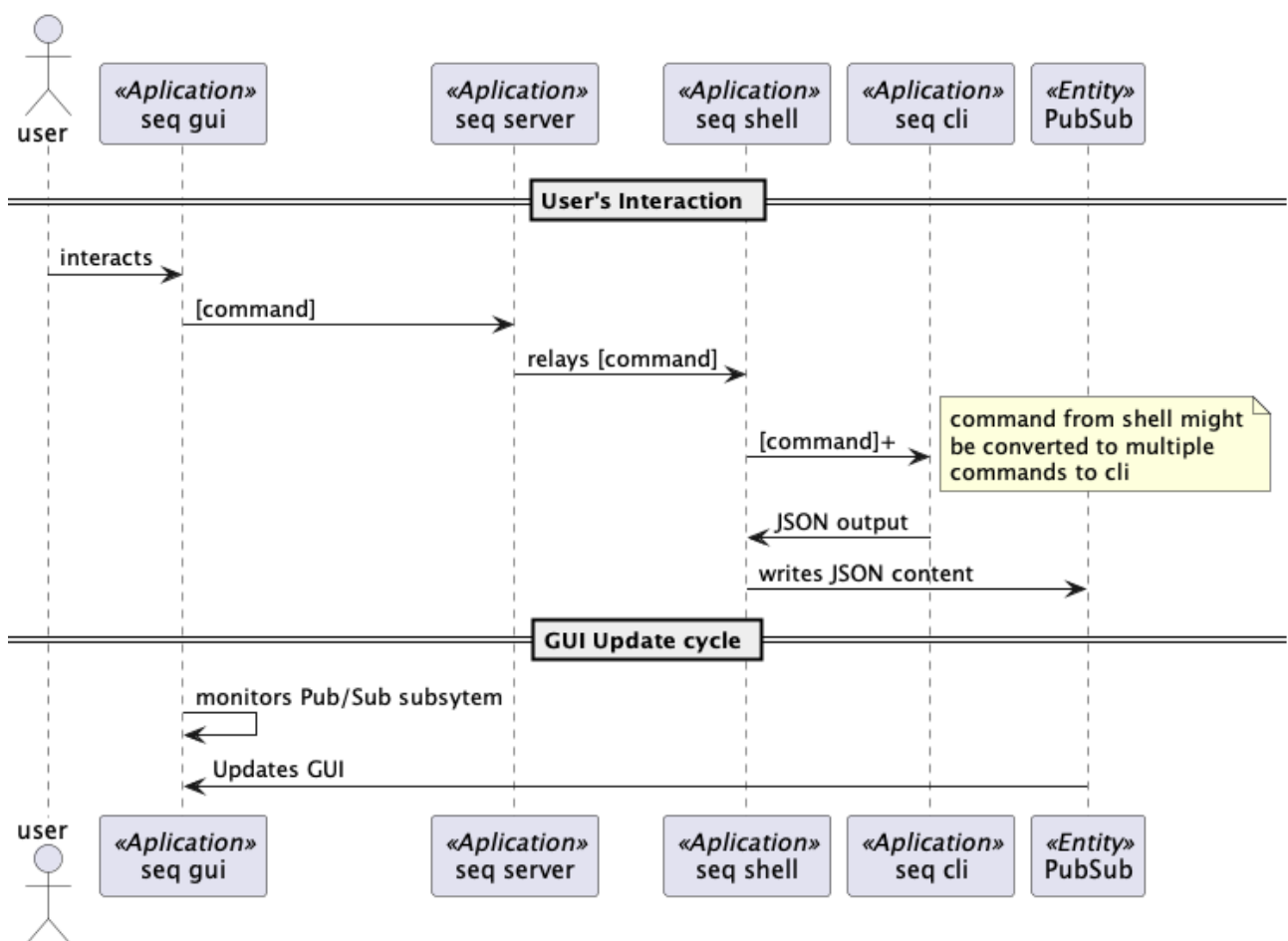


Fig. 1: GUI interaction with server.



```
$ seqtool gui --address 8000
```

4.2 Using the Sequencer GUI

Starting the sequencer GUI is done as follows:

```
$ seqtool gui
```

One can pass change the defaults values for hosts and ports using the following options.:

```
$ seqtool gui --help
Usage: seqtool gui [OPTIONS] [SCRIPT]...

Client application for the Sequencer that allows to load OBs and execute
scripts and monitor their progress.

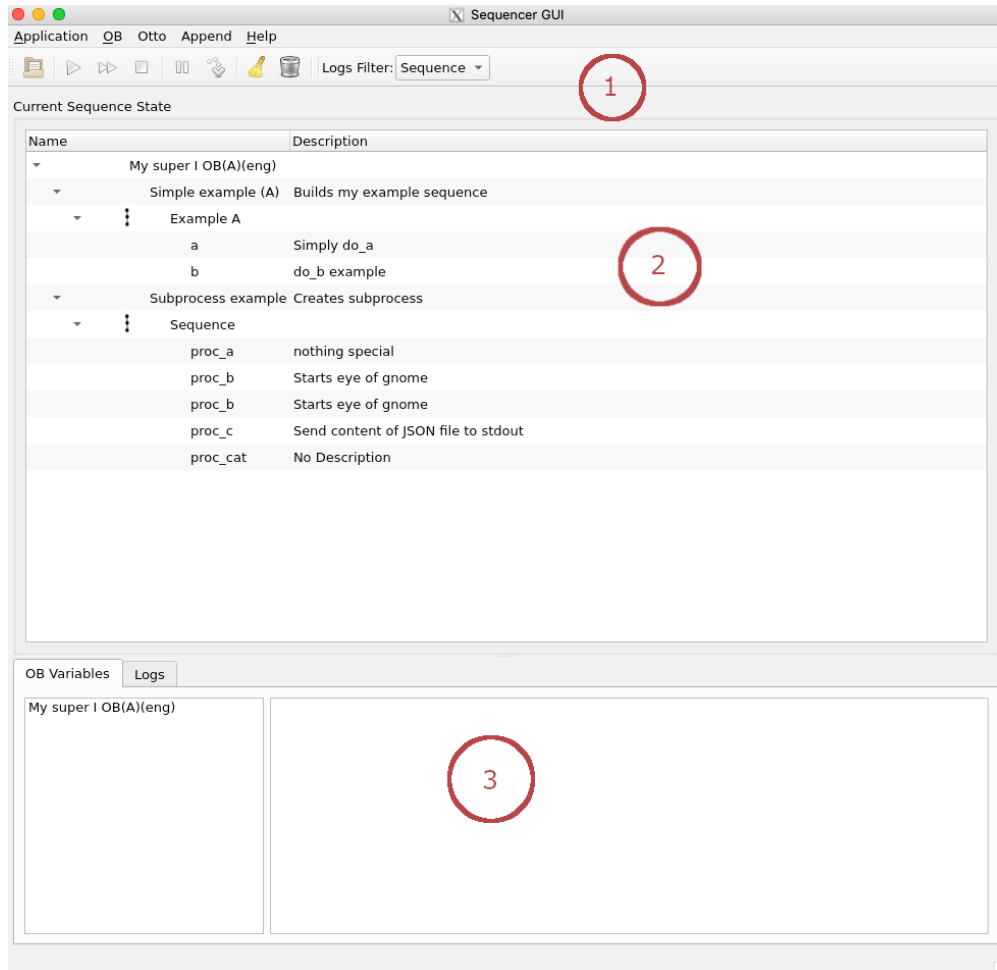
Options:
  --address TEXT          Sequencer server address [HOST:]PORT
                          [default: 8000]
  --log-level [DEBUG|INFO|WARN|CRITICAL|ERROR]
                          Will present logs at this level of higher
                          [default: INFO]
  --run                   Run the script passed as parameter
  --config TEXT           sequencer configuration
  --help                  Show this message and exit.
```

The defaults values used by the sequencer server and the sequencer gui will suffice to run all of this in the same machine.

If any port clash is foreseen, please use the command line options to change port values to suitable ones.

It is possible to automatically start a script when launching the gui with the `--run` option or to specify a configuration file with `--config` (see *Configuration File*).

The GUI opens the following window.



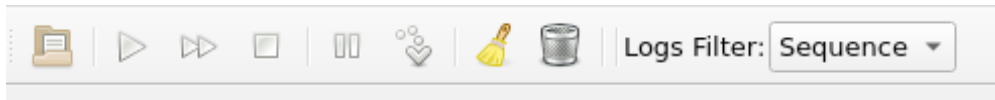
The GUI will try to automatically connect on start to the sequencer server, and its logs.

The GUI is divided in three parts.

1. The toolbar, displays button to open Observing Blocks, manipulate the OB, its steps, and filter logs.
2. The Sequence tree. For each sequence step, displays two columns: name, and a description in case provided.
3. The logs and OB variables tabs:
 - OB variables presents and allows edition of variables for the block.
 - Logs shows entries belonging to the Observing Block, and its steps, that are logged using **python logging module**.



The Toolbar



The GUI toolbar sports the following buttons and options.

Append OB

Loads and append to the sequencer server the selected OB in json format. When pressed, a file selection dialog will appear. The user may select an OB file.

Run

Executes the scripts loaded into the sequencer.

Continue

A script(s) execution can be aborted or paused. The execution resumes with the *Continue* button.

Abort

Stop a script that is currently in execution.

Reset

Cleans the execution tree and restarts the execution engine.

Pause

Sets the selected node to pause when reached. The same button may be used to unpause a step.

Skip

Marks the selected node to be skipped when reached. The same button may be used to toggle this option off.

Logs Filter

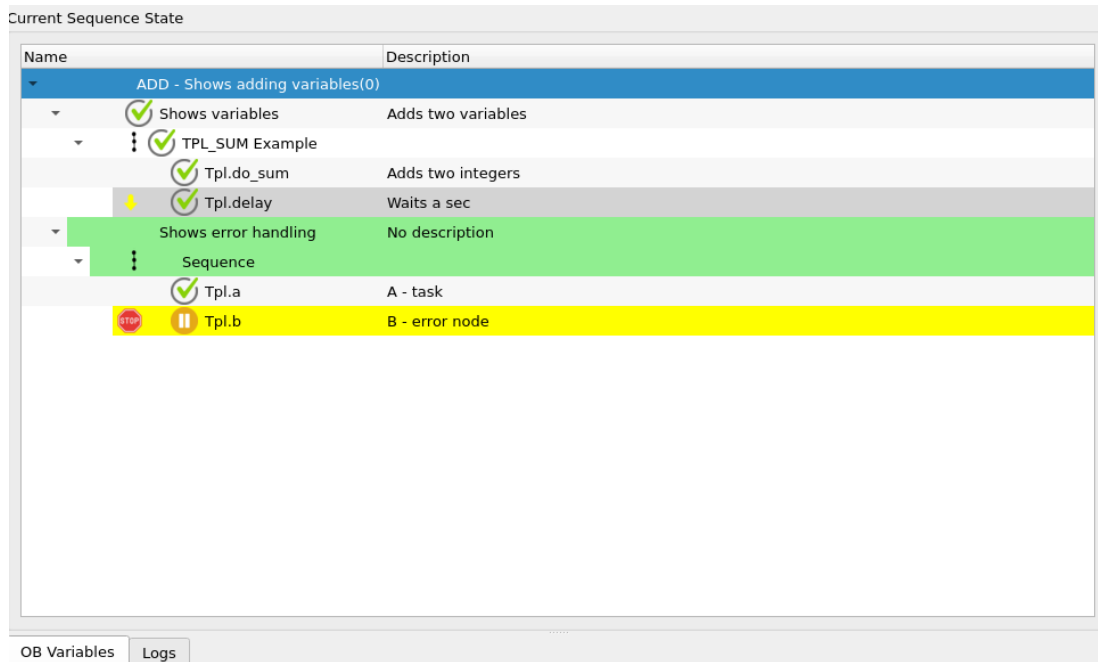
This dropbox applies a filter to the logs (section 3 in the GUI).

- Sequence: shows logs produced by the explicit calls to the logger by the author of the sequence.
- seq.lib: shows internal logs produced by the sequencer.
- seq.user: shows logs that describes the changes in state of the sequence.

The toolbar commands can also be given directly on the input widget.






The Tree Window



The tree window shows the loaded scripts in a tree widget. The script and its components can be expanded or collapsed. Each column shows a single node with its name, state and doc string (from python).

Note: The mouse secondary button displays a context menu that allows to *Pause/UnPause* or *Skip/UnSkip* the selected node.

Along the node number an icon is displayed, which depends on the node type to help identify their intent.

-  Marks Sequence nodes.
-  Marks Parallel nodes.
-  Marks Loop nodes.

When the sequence starts (by pressing the Run Button on the toolbar), this view will automatically scroll to the latest node with Running state. Also, they will be highlighted in light green color while the node is in Running state.



Logs

OB Variables			
Logs			
Level	Timestamp	Logger	Message
INFO	2023-04-01 05:05:07,947.947	seq.user	Starting Task: Tpl.do_sum
INFO	2023-04-01 05:05:07,947.947	seq.samples.t...	xSUM 1 + 2 = 3
INFO	2023-04-01 05:05:08,950.950	seq.user	End Task: Tpl.do_sum
INFO	2023-04-01 05:05:08,960.960	seq.user	Starting Task: Tpl.a
INFO	2023-04-01 05:05:08,960.960	seq.samples.t...	A
INFO	2023-04-01 05:05:08,960.960	seq.user	End Task: Tpl.a

This view presents a table with logs produced by the sequencer. The table contains a row for each log, with the following columns:

- Level**
The log type, or level, which can be DEBUG, INFO, ERROR, CRITICAL, EXCEPTION and FATAL. The colors helps to state the importance of the log entries.
- Timestamp**
Date and time of the log entry, up to milliseconds, using the sequencer server reported time.
- Logger**
Name of the logger used to produce the entry. For developers of sequences is important to use names that do not start with *seq.lib*.
- Message**
The log message displayed.

OB Variables

OB Variables			
Logs			
My humble II OB(A)	Step/Variable	Data Type	Value
My humble II OB(A)	0		
My humble II OB(A)	par_b	integer	0
My super I OB(A)	par_c	number	77
	1		
	par_b	integer	0
	par_c	number	10

Set Variables on Sequencer

Variables for Observing Blocks can be set through this Tab.

The left side of the Tab shows the different OBs loaded in the sequencer server, while the right side display the variables. Variables are grouped by the node in which they are required, and display the



name of the variable, the data type expected, and its current value.

Double clicking on a particular value edits the existing value.

Once a user is satisfied with the changes, they can be applied back to the sequencer server by using *Set Variables on Sequencer* Button.

Note: Variables are saved back to the server per OB, so if changes were made for more than one OB, the user needs to select that particular OB, and press the *Set Variables on Sequencer*.

OTTO Interface

For an explanation about OTTO's purpose see *Interface to Observation Handling SW*. The GUI allows to fetch and execute OBs from a VES (visitor execution sequence). The OTTO pop-up menu is enabled if the configuration defines OTTO settings and consists of the following actions. See *Otto popup menu*.

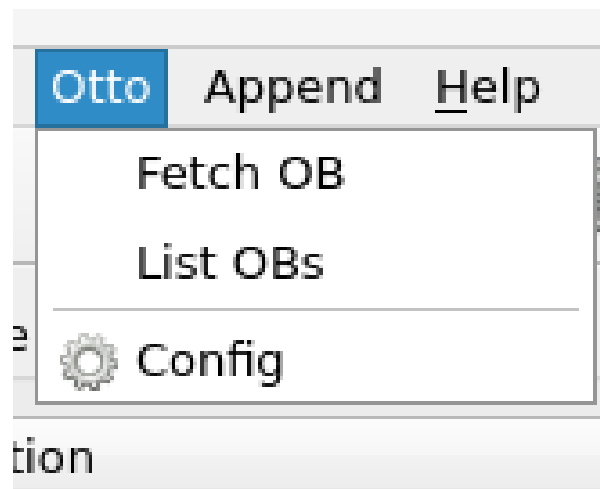


Fig. 2: Otto popup menu.

Fetch OB

Fetch next execution sequence from the queue.

List OBs

List OBs in the queue. See *OTTO list available OBs*.

Config

Opens OTTO server configuration dialog.

OTTO's default configuration is found in the GUI's configuration file but the GUI allows to configure its parameters as shown in *OTTO server configuration dialog*

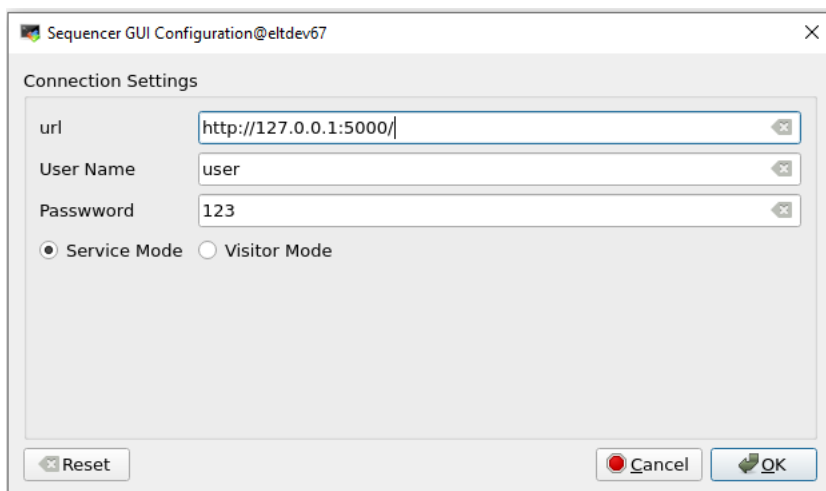


Fig. 3: OTTO server configuration dialog

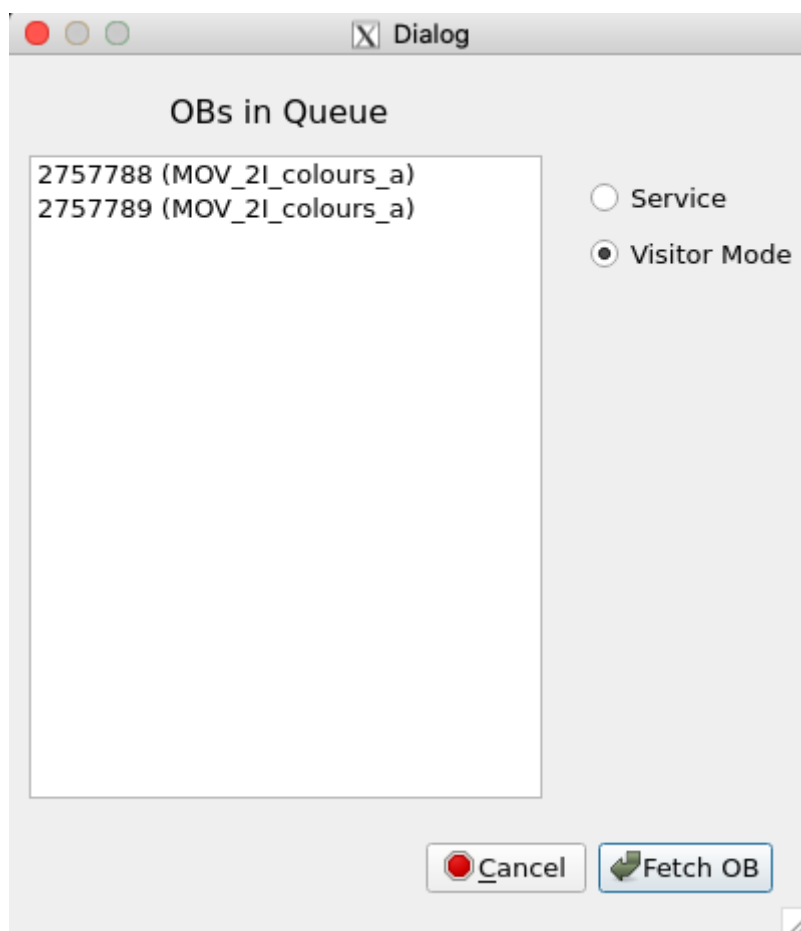


Fig. 4: OTTO list available OBs



Error Handling

When a script aborts with an error (some exception was raised) a dialog window appears and displays the python traceback.

The dialog presents three buttons that allows to:

1. *Retry* the failed node.
2. *Ignore* the error and resume script execution.
3. *Discard* the error window and reflect on what just happened.



In addition, an aborted script can be *continued* (skipping the failed node) by means of the *continue* button in the toolbar.

One can retry the failed node typing the *retry* command and using the node serial number.



Advanced: Debug Mode

The *Application* Menu hold the *Debug Mode* option. When clicked, it will present more options oriented at developers of sequences.

The screenshot shows the Sequencer GUI with the 'Current Sequence State' table and the 'Console' log.

Name	#	State	Description
└ TPL_B	10	Not Started	
└ Tpl.do_some	11	Not Started	Adds par_b and par_c
└ Template	12	Not Started	
└ TPL_B	13	Not Started (Pause)	
└ Tpl.do_some	14	Not Started (Skip)	Adds par_b and par_c
└ My humble II OB(A)	15	Not Started	
└ Template	16	Not Started	
└ TPL_B	17	Not Started	
└ Tpl.do_some	18	Not Started	Adds par_b and par_c
└ Template	19	Not Started	
└ TPL_B	20	Not Started	
└ Tpl.do_some	21	Not Started	Adds par_b and par_c
└ My super I OB(A)	22	Not Started	
└ Template	23	Not Started	
└ Sequence	24	Not Started	
└ a	25	Not Started	Simply do_a
└ b	26	Not Started	do_b example
└ Template	27	Not Started	
└ Sequence	28	Not Started	
└ proc_a	29	Not Started	nothing special
└ proc_b	30	Not Started	Starts eye of gnome
└ proc_b	31	Not Started	Starts eye of gnome
└ proc_c	32	Not Started	Send content of JSON file to stdout
└ proc_cat	33	Not Started	No Description

The Console log shows the following messages:

```
12:15:48.731 sent: ob /home/eltdev/repos/seq/tpl2.json
12:15:48.745 received: ob /home/eltdev/repos/seq/tpl2.json
12:15:50.069 sent: ob /home/eltdev/repos/seq/tpl2.json
12:15:50.101 received: ob /home/eltdev/repos/seq/tpl2.json
12:15:51.288 sent: ob /home/eltdev/repos/seq/tpl1.json
12:15:51.317 received: ob /home/eltdev/repos/seq/tpl1.json
12:15:59.755 sent: help
12:15:59.757 received: help
12:16:10.291 sent: flip pause 13
12:16:10.293 received: flip pause 13
12:16:22.891 sent: flip skip 14
12:16:22.894 received: flip skip 14
```

The Nodes Treeview in section 2 of the GUI will display two more column: the internal serial number assigned to the step (column #) and the *state* column.

The *state* column displays not only the node state but also if the node has been marked to pause or to skip its execution. Pause and Skip are runtime flags, which extend the description of the state for a given node. Node states can be one of the following:



Not Started

Node has not been started yet.

Scheduled

Script execution started. The node will be executed at some point.

Running

Node is currently executing.

Finished

Node has completed its execution. A *finished* node can be in any of the following substates:

Skip

Node is considered finished because it was purposely skipped (Skip runtime flag).

Error

The node raised a runtime exception and has finished with error.

Cancelled

Node execution has been cancelled. This happens when some other node, down the tree has finished with error.

Paused

The execution of the tree is paused in the node mark with Pause runtime flag. A paused script can be resumed by removing the Pause flag from the node. This is done with the *right mouse button menu* and select *Pause/UnPause* node, or by selected the node, and using the Pause Button in the toolbar.

The Tabs in section 3 of the GUI will display one more Tab named *Console*

The console is a direct communication line with the sequencer server, in which the user can execute commands. This is not intended for final users and commands entered here will not display its output on the same console.

It accepts the same commands for the sequencer kernel or shell. For feedback please see the standard output on the sequencer server.

4.3 Configuration File

The Sequencer GUI reads the configuration upon startup from the file specified with the `--config` option. The path can be either absolute or relative to a folder defined in the `CFGPATH` environment variable. It consists of two sections, one for the Sequencer server, the other for OTTO configuration (optional). Below an example of a configuration file:

```
seq_server:
  url: pl3.pl.eso.org:8000
  loglevel: INFO
```

(continues on next page)



(continued from previous page)

```
otto:
url: http://127.0.0.1:5000/
insid: FORS2
mode: VM
```

On the `seq_server` section the default `HOST:PORT` for the server is specified. However this value is just an initial guess. The GUI will attempt to spawn a server on the specified URL. However if the port is already in use it will increase the port number up to 10 times until a free port is found and the server is started there.

The OTTO section provides the default OTTO server url and the instrument.

If no configuration file is provided, the default configuration will set the server to `hostname:8000` and `loglevel` to `INFO`.

4.4 Sequencer server

Since the Sequencer GUI takes care of starting and stopping the *sequencer server* there should be no need for an user to start on its own. Anyway, this is how one can start the sequencer server and its options:

```
$ seqtool server --help
Usage: seqtool server [OPTIONS]

Starts the Sequencer Server, a socket server for the sequencer.

It listens on the "address" for commands and applies them to its own
instance of sequencer (seqtool shell).

If you need multiple instances of the sequencer server, please start them in
their own port and provide their own instance of redis.

Options:
  --address TEXT           Sequencer server address [HOST:]PORT
                           [default: 8000]
  --log-level [DEBUG|INFO|WARN|CRITICAL|ERROR]
                           Will present logs at this level of higher
                           [default: INFO]
  --help                  Show this message and exit.
```

The sequencer server can be executed on a different host than the GUI. In that case, the server address is given as `[HOST:]PORT`.



5 A Deeper Look

A deeper look to nodes and its attributes is given in the following paragraphs.

5.1 Passing Arguments to Actions

Action and ActionInThread constructors only admits a function object, no space to pass parameters to the function or coroutine that is going to be executed by the node.

However, *functions* are first class objects in python, this makes easy to create them and pass around dynamically. It is recommended to use *partial* functions in order to pass parameters to the functions associated to action nodes.

5.2 Using partial functions

The use of *partial functions* allows to fix a certain number of arguments of a function and generate a **new function**, on the spot. Please see `functools.partial()` for the official documentation. In any case the following example illustrates its use.

We recommend the use `seq.lib.partial()` instead of `functools.partial()` since the former will keep the documentation of the wrapped function.



Listing 8: partial function example

```
from seq.lib import partial

def f(a,b,c):
    return a+b+c

g = partial(f,1,2) # creates new function g()
g(3) # Equivalent to f(1,2,3)
g(9)
g(1,2) # f will complain too many arguments were passed
```

The example *partial function example* shows a new function *g()* created by using *partial* in order to specify that *g()* when, called will invoke *f(1,2)* plus any extra argument given.

From the point of view of the python interpreter *g()* is a normal function, meaning you can use *g()* as many times as you see fit.

5.3 Runtime Flags

It is possible to associate *runtime flags* to the nodes. In order to *skip* or *pause* them. The easiest way to do this is by using the *_pause* and *_skip* shortcuts, as follows:

```
from seq.lib import _pause, _skip, Sequence

async def a():
    pass

def b():
    pass

node1 = _pause(a);
node2 = _skip(b);

s = Sequence.create(node1, node2, _skip(a))
```

Another way is to use *set_runtime_flags(node, flags)* function which requires a node and sets its runtime flags from the parameter. Valid runtime flags values are *PAUSE* and *SKIP*:

```
from seq.lib import RTFLAG, set_runtime_flag

node = Action(f)
set_runtime_flag(node, RTFLAG.PAUSE)
...
set_runtime_flag(node, RTFLAG.SKIP, False)
```



5.4 Summary building DAGs

Constructor Calling conventions

First of all, except for `Action` and `ActionInThread` do not use the standard `class_name()` constructor to build nodes. i.e. never use naked `Parallel()` to create a `Parallel` node, same for the other node types.

Important: Use the `create()` method to instance all container classes (`Loop`, `Sequence`, `Parallel`, etc).

Since they hold a variable number of children nodes, their constructors (**`create()`** method) receives its children as positional arguments. Any other attribute (mandatory or not) is passed through keyword arguments. See *Node constructor calling convention*

Listing 9: Node constructor calling convention

```
# Sequence ctor
Sequence.create(child_1, child_2, ..., child_n,
                id="my_unique_id", name="my nice name")

# Parallel ctor
Parallel.create(child_1, child_2, ..., child_n,
               id="my_unique_id", name="my nice name")

# Loop ctor
Loop.create(child_1, child_2, ..., child_n,
            id="my_unique_id", name="my nice name",
            init = init_function,
            condition= condition_function
            )
```

Node Attributes

Node attributes are passed as keyword arguments in its `create` method. Every node has, at least, a `name` and `id` attributes. If they are not specified while building the DAG they are assigned by the engine pseudo-randomly.

The specific attributes for each node type are detailed in the following table:

Node Class	Attribute	Description
<i>ALL</i>	<code>id</code>	Node id (must be unique)
<i>ALL</i>	<code>name</code>	Node name
<code>Loop</code>	<code>init</code>	Initialization node
<code>Loop</code>	<code>condition</code>	Loop's condition node



5.5 Special Variables

Sequences uses `contextvars.ContextVar` to distribute special values around all tasks and threads that are part of a given Sequencer script.

The *ContextVar* module implements **context variables**. This concept is similar to thread-local storage (TLS), but, unlike TLS, it also allows correctly keeping track of values per asynchronous task, e.g. `asyncio.Task`.

Class	Variable	Description
Sequence	<code>current_seq</code>	Current Sequence name
Sequence	<code>root</code>	The root node
Loop	<code>index</code>	Loop's running index

5.6 Result Handling

Each node has a **result** attribute where the return value from its associated step is stored.

For *Action* and *ActionInThread*, the result is just the return value from its associated function. Since each *Action* is just a simple function or method, their result is kept in its corresponding node's attribute *result*.

All other nodes will have an empty result unless it is explicitly set by some step, in the sequence. Example *Set node result* shows a step setting the result of the sequence that contains it. The node that contains a given action is accessed through the *Sequence.current_seq* context variable as the example shows.

It is clear that in order to check for a node's result one needs to have a handler to that node or a way to find it, see *Finding nodes*.

Note: Both *Parallel* and *Loop* classes inherit from *Sequence*. Therefore they can access *Sequence.current_seq* context variable.

Listing 10: Set node result

```
from seqlib.ob import OB
# reuse some sequence ...
from . import test_a as a

async def mystep():
    """Sets current Sequence's result"""
    s = Sequence.current_seq.get()
    s.result = 0
```

(continues on next page)



(continued from previous page)

```
async def test_result():
    tpa = OB.create_sequence(a)
    sc = Parallel.create(tpa, mystep)
    await sc.start()
    assert sc.result == 0
```

5.7 Finding nodes

Unless one has saved a reference to a node, e.g. as a class member or global variable. The only way to find a node in the DAG is through its *unique id*.

The func: `seq.lib.nodes.find_node` receives as a parameter a starting node and the *id* of the node being looked up. On success it returns a tuple the target node and its parent as the tuple (*parent*, *node*).

1. In order to lookup a node from the DAG's root (meaning look around the complete sequence until a hit is found), simply pass the *root* context variable as the initial node.
2. In order to lookup a node from the *current sequence* use the *current_seq* context variable as the starting node. *Lookup a node* illustrates both use cases.

Listing 11: Lookup a node

```
from seqlib.nodes import Sequence, find_node
...
async def do_a():
    # find node `id1` (not shown) starting at root and get its result.
    _, node = find_node(Sequence.root.get(), "id1")
    print("the other node result", node.result)
    return node.result + 1; # or something

async def do_b():
    # find do_a's result
    _, node_a = find_node(Sequence.current_seq.get(), "id_do_a")
    return node_a.result + 1; # or do something else

# Given a Sequence s
s = Sequence.create(Action(do_a, id="id_do_a"), do_b)
```

Since *node_ids* are unique inside a given *Sequence* there is no risk of losing an *Action's* result because it gets overwritten by some other node. As opposed to *Nodes have context*.



5.8 Nodes have context

Besides the *result* attribute that can be inspected in order to pass information between Sequencer scripts. There is the *context* dictionary which can be freely accessed throughout all nodes being executed.

The context dictionary is a property shared among all Sequence nodes (includes Loop and Parallel). Action and ActionInThread nodes can gain access to it through `Sequence.get_context()` static method. Please see the following code excerpts *Node context example*.

The methods `do_a()` and `do_b()` must access the context through the `Sequence.get_context()` static method. The object `tpl`, being a Sequence instance can access its `Sequence.context` attribute.

Listing 12: Node context example

```
async def do_a():
    ctx = Sequence.get_context()
    ctx["do_a"] = 1

def do_b():
    ctx = Sequence.get_context()
    ctx["do_b"] = 1

# creates the sequence
tpl = Sequence.create(do_a, ActionInThread(do_b))
# Sequence.root.set(tpl)
await tpl.start()
ctx = tpl.context

assert tpl.context["do_a"] == 1
assert tpl.context["do_b"] == 1
```

Warning: Notice there are no hard rules about what can go into the context dictionary and under what key. It might be simpler to use than setting results on nodes but there is no guarantee a given key might be overwritten in some other part of the running script just because of a name clash.



5.9 Node Types

The sequencer node types lives in the module `seq.nodes`:

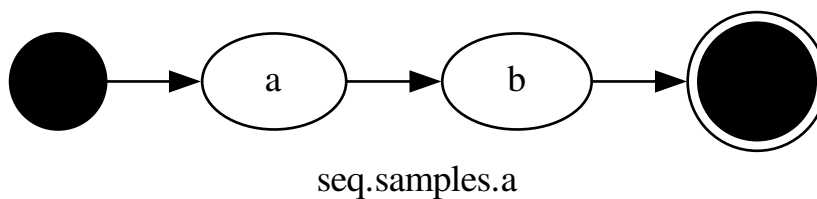
Action

The simple action node. It contains a python function or method to be executed.

```
# Creates a node with some properties.  
node_a = Node(t.a, name="node")
```

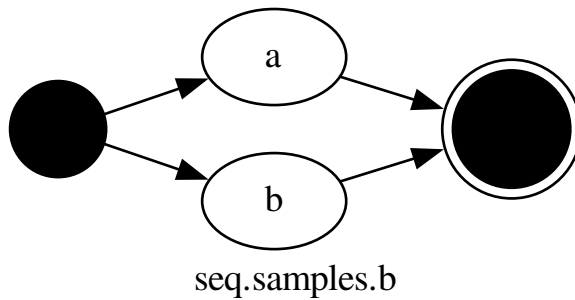
Sequence

Executes nodes one after the other.



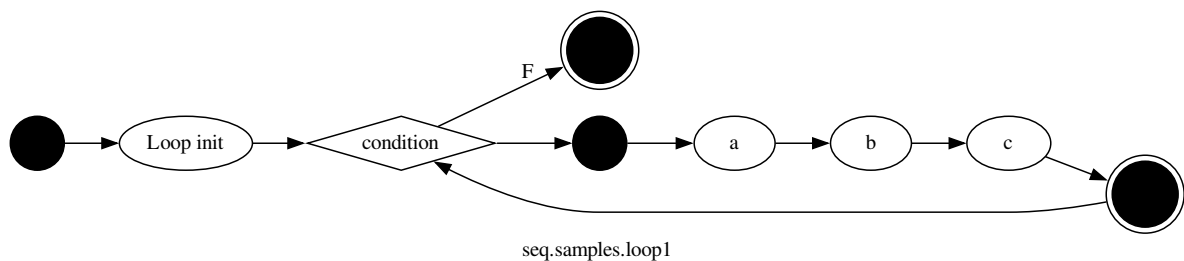
Parallel

Executes nodes in parallel, finishes when all its nodes are done.



Loop

A *Loop* consists of a *condition* and a *block* of nodes to execute while said conditions remains True. It also accepts an initialization function to be called, only once, before the first loop iteration.



```
"""
Implements a loop.
The condition checks Loop's index < 3.
"""

import asyncio
import logging
import random
from seq.lib.nodes import Loop

logger = logging.getLogger(__name__)
```

(continues on next page)



(continued from previous page)

```
class Tpl: # Mandatory class name
    async def a(self):
        """sleeps up to 1 second"""
        t = random.random() # 0..1
        await asyncio.sleep(t)
        logger.info(".. done A: %d", Loop.index.get())

    async def b(self):
        """sleeps up to 1 second"""
        t = random.random() # 0..1
        await asyncio.sleep(t)
        logger.info(".. done B: %d", Loop.index.get())

    async def c(self):
        pass

    async def check_condition(self):
        """
        The magic of contextvars in asyncio
        Loop.index is local to each asyncio task
        """
        logger.info("Loop index: %d", Loop.index.get())
        return Loop.index.get() < 3

    @staticmethod
    def create(**kw):
        t = Tpl()
        l = Loop.create(t.a, t.b, t.c,
                       condition=t.check_condition, **kw)

        return l
```



6 Good Practices

Here we provide some advice on how to use the Sequencer API.

6.1 Writing Sequences

Code Structure

For short scripts, putting together a Sequencer script out of a handful of functions or coroutines is perfectly fine. OTOH, in order to provide reusable code, it is preferable to group common functionality using classes.

Logging

The sequencer uses the standard python logging module `logging` and provides the logger `seq.user`.

User's logging to a file (`seq_user.log`), its exact location depends on the tool used. See *Sequencer Command Line Tools*.

The `seq.lib.getUserLogger()` function returns a standard logger object.

Listing 13: Logging support

```
from seq.lib.log import getUserLogger
user_logger = getUserLogger()

def my_method():
    user_logger.info("My INFO msg")
```

Creating Sequences

Follow the convention outlined in *Constructor Calling conventions*. The class' `create()` method or module's `create_sequence()` should get its required arguments as *positional arguments* or variable list of arguments (`*args`). Customization shall be accepted through *Keyword Arguments* (`**kw`) in order to allow the user to setup, at least, the nodes' *name* and *id*.

The following example shows a module level sequence ctor (`create_sequence`), It creates a new Sequence out of the arguments passed,

The returned sequence has an extra final step `my_end_step()` that was added by the constructor. The usage of *keyword arguments* allows to pass options to the underlying *Sequence* object.



Listing 14: Implement constructor convention

```
# module ctor
async def my_end_step():
    pass

def create_sequence(*args, **kw):
    return Sequence.create(*args, my_end_step, **kw)
```

Listing 15: class example

```
# Class Example
# must be named `Tpl`
class Tpl:
    async def my_end_step():
        pass

    @staticmethod
    create(*args, **kw):
        p = Tpl()
        return Sequence.create(*args, p.my_end_step, **kw)
```

Invoke other processes

It is important not to redirect the or change *stdin* or *stdout* of the sequencer tools.

To invoke other processes, and keep them detached from the sequencer process use *asyncio.create_subprocess_shell()* and pass options *DEVNULL* for *stdin* and *stdout* as shown in the example.

Listing 16: subprocess example

```
from seq.lib.nodes import Sequence
import asyncio

async def proc_a():
    print("A")

async def proc_b():
    await asyncio.create_subprocess_shell('eog', stdin=asyncio.subprocess.DEVNULL,
                                          stdout=asyncio.subprocess.DEVNULL)

def create_sequence(*args, **kw):
    return Sequence.create(proc_a, proc_b)
```



7 Sequencer Command Line Tools

7.1 The *seqtool* meta command

The *seqtool* provides the following subcommands:

run Allows to execute a sequencer script.

draw Generates DAG images of Sequencer scripts.

shell

Starts an interactive CLI which allows to load and run sequencer scripts.

gui

Starts the sequencer GUI

server

Starts the sequencer server

Common command-line options

Every *seqtool* sub-commands supports the following options:

-help

Shows command's syntax and usage options.

-log-level

Sets the process' logging level.

The *seqtool shell* sub command

Starts an interactive CLI where the user can submit commands to the sequencer library in order to execute Sequencer scripts.

Supported commands are:

help

Provides a list of commands supported by the CLI. With a parameter, displays the documentation of the given command. e.g.:

```
(seq)>> help load
load command
Will display `load`'s command documentation
```

quit

Stops the *shell* process.

load



Loads a python module that implements a sequencer script. The module has to be specified as Python would import it. e.g.:

```
(seq)>> load seq.samples.a
```

modules

Lists the modules loaded by the Sequencer core.

run

Executes *all* the Sequencer scripts loaded.

tree

Shows the Sequencer tree, from the modules loaded. Along with the tree structure it displays the node's serial number needed by some commands.

pause <node_sn>

Allows to mark a node to pause execution. Receives the node serial number as parameter.

Alternate syntax with node name.

skip -find_name <node_name> -no-flag

Allows to mark a node to pause execution. Receives the node name as parameter. if -no-flag is passed the node is mark to NOT PAUSE.

resume <node_sn>

A paused script can be resumed with this command. One has to give the node serial number to resume from (the *PAUSED* node).

skip <node_sn>

Allows to mark a node to skip from execution. Receives the node serial number as parameter.

Alternate syntax with node names. **skip -find_name <node_name> -no-flag**

Allows to mark a node to skip from execution. Receives the node name as parameter.

if -no-flag is passed the node is mark to NOT SKIP.

retry

Allows to retry the execution of a failed node.

continue

When the execution of a script is cancelled, due to an error. One can resume execution from the next available node with the *continue* command.

flip <skip|pause> <node_sn>

Flips the *pause* or *skip* flag of a node. Must give the flag to flip and the node's serial number:



```
(seq)>> flip pause 3  
(seq)>> flip skip 4
```

bp <command>

bypass to subprocess command

save

save the current session

break line

set a break at the specified line

err

forces an exception

lsob

list loaded obs

nodes

execute the obs' nodes

session

load session sequences

unskip line

skip specified line

clear

kill seq-core subprocess

fetch

fetch an OB from OTTO

init

init command, creates sequencer exec process

lsvar

list variables

ob <obpath>

loads an OB file (json)

reset

executes all loaded modules

Upon start, a *session directory* is created, below the */tmp* directory as *seq_session_<pid>*. The following log files can be found there:

seq_user.log

User's logging and very basic script execution info.

seq.log

Internal shell's logging. Useful for debugging the sequencer itself.



kernel.log

Internal sequencer logging, state changes, execution chain, etc. Useful for debugging the sequencer itself.

Basic Usage

Command line options

The *seqtool server* command accepts the following command line options:

- *-address* (as [HOST:]PORT) where to listen for connections
- *-redis* (as [HOST:]PORT) where is the REDIS server

Loading sequences

At the prompt one can send commands to the shell. The following will load a sequence module (if found):

```
$ seqtool shell
MAIN ...
INFO:(seqsh.__init__): shell started: /tmp/seq_session_13369/seq.log
(seq)>> INFO:(seqsh.do_init): initialize shell- False
INFO:(seqsh.do_init): dummy redis connection
INFO:(seqsh.start_seq_core): Starting seq kernel
INFO:(seqsh.start_seq_core): connected to child process

(seq)>> load seq.samples.a
```

One can examine the structure and state of the loaded script with the *nodes* command:

```
(seq)>> nodes
(seq)>> (Core)> A-- (5) begin NOT_STARTED
A-- (8) end NOT_STARTED
S+- (2) Sequence NOT_STARTED
  A-- (6) begin NOT_STARTED
    A-- (3) a NOT_STARTED
    A-- (4) b NOT_STARTED
    A-- (7) end NOT_STARTED
```

To decipher the above output, every line is formatted as:

<Node type> – (<Node_number>) <Node name> <state|flags>

Lines are indented according to its level in the DAG. Every node in the DAG is prefixed by its type, as follows:



A

This for *Actions*.

S

Refers to a *Sequence*.

P

Indicates a *Parallel* sequence.

L

To indicate a *Loop*.

The number in parenthesis are the node's serial number which some commands needs as a parameter.

Sequencer scripts are executed with the *run* command.

Executing Sequences

The *run* command executes the loaded sequences:

```
seq)>> run
(seq)>> INFO:seq.samples.a:a
INFO:seq.samples.a:B
```

The state of the script can be observed at any time with the *nodes* command:

```
A-- (5) begin FINISHED
A-- (8) end FINISHED
S+- (2) Sequence FINISHED
  A-- (6) begin FINISHED
  A-- (3) a FINISHED
  A-- (4) b FINISHED
  A-- (7) end FINISHED
```

Skipping Nodes

A node can be skipped from execution with the *skip* command and indicating the node number one desires to skip

```
(seq)>> skip 3
(seq)>> run
INFO:seq.samples.a:B
```

The nodes command after running the sequence looks as follows:



```
(seq)>> nodes
(Core)> A-- (5) begin FINISHED
A-- (8) end FINISHED
S+- (2) Sequence FINISHED
  A-- (6) begin FINISHED
  A-- (3) a FINISHED|SKIP|RT.SKIP
  A-- (4) b FINISHED
  A-- (7) end FINISHED
```

Node (3) is marked as *FINISHED* and *SKIP* (was skipped). The *RT.SKIP* text indicates the *runtime flag SKIP* is active.

In the example above all nodes are *FINISHED*, no error was detected.

Pause a script

The *pause* commands requires the node number where to pause. Pausing on node (4) on the script we are working on activates the *pause runtime flag* on said node:

```
(seq)>> pause 4
(seq)>> nodes
(Core)> A-- (5) begin FINISHED
A-- (8) end FINISHED
S+- (2) Sequence FINISHED
  A-- (6) begin FINISHED
  A-- (3) a FINISHED|SKIP|RT.SKIP
  A-- (4) b FINISHED|RT.PAUSE
  A-- (7) end FINISHED
```

Executing the script will pause on node (4). It will look as follows (nodes command output redacted)

```
A-- (4) b PAUSED|RT.PAUSE
```

In order to resume execution just issue the *resume* command, with the node number:

```
(seq)>> resume 4
INFO:seq.samples.a:B
(seq)>> nodes
(Core)> A-- (5) begin FINISHED
A-- (8) end FINISHED
S+- (2) Sequence FINISHED
  A-- (6) begin FINISHED
  A-- (3) a FINISHED|SKIP|RT.SKIP
  A-- (4) b FINISHED|RT.PAUSE
```

(continues on next page)



(continued from previous page)

```
A-- (7) end FINISHED
(seq)>>
```

To clean the runtime flags use *flip* command, in the example nodes (3) and (4) are back to normal:

```
(seq)>> flip skip 3
(seq)>> flip pause 4
```

Error Handling

When a *Sequencer Item* generates an error, i.e. raises a Python exception, then the whole sequence is *CANCELLED*, and the offending node is marked as *FINISHED|ERROR*. The following example shows such a sequence:

```
(seq)>>
(seq)>> load seq.samples.test_err
(seq)>> (Core)> NODES: ['begin__seq_ROOT__qVJL9XVXR3', 'Sequence_3j7VG0YWmM',
↳ 'end__seq_ROOT__qVJL9XVXR3']
NODES: ['begin_Sequence_3j7VG0YWmM', 'Tpl.a_3jxvx', 'b', 'Tpl.c_1wWmj', 'end_Sequence_
↳ 3j7VG0YWmM']

(seq)>> run
(seq)>> SEQRUN!, catch exception: Sequence

.....

File "/home/eeltdev/introot/lib/python3.7/site-packages/seq/samples/test_err.py", line 28, in b
    x = 1 / 0
ZeroDivisionError: division by zero
```

The exception stacktrace is shown clearly indicating the offending method and line where the exception occurs (division by zero). The output of *nodes* command:

```
(seq)>> nodes
(seq)>> (Core)> A-- (7) begin FINISHED
A-- (10) end CANCELLED
S+- (6) Sequence CANCELLED|ERROR
    A-- (8) begin FINISHED
    A-- (3) Tpl.a FINISHED
    A-- (4) Tpl.b FINISHED|ERROR
    A-- (5) Tpl.c CANCELLED
    A-- (9) end CANCELLED
```

Where the node *b* has state *FINISHED|ERROR*.



One the can *retry* the failed node or *continue* from the next unfinished node.

7.2 seqtool run

Executes the Sequencer scripts given in the command line. Usage:

```
$ seqtool run --help
```

Starts seq CLI

Usage: seqtool run [OPTIONS] [MODULES]...

Options:

--log-level [DEBUG|INFO|WARN|CRITICAL|ERROR]

Will present logs at this level of higher
[default: INFO]

--help Show this message and exit.

It is non-interactive so it breaks at the slightest provocation. An example:

```
# specify two modules to execute in sequence
$ seqtool run seq.samples.a seq.samples.b
```

log files

The *seqtool run* commands creates the following log files at user's current directory.

seq_user.log

User's logging and very basic script execution info.

seqrun_<pid>.log

Internal sequencer logging, state changes, execution chain, etc. Useful for debugging the sequencer itself.

7.3 seqtool draw

Draws the graph representation of the Sequencer scripts given in the command line. Usage:

```
$ seqtool draw --help
```

Usage: seqtool draw [OPTIONS] OUTPUT [MODULES]...

Draws the graph representation of a Sequencer script given in the **command** line.

(continues on next page)



(continued from previous page)

Options:

```
--log-level [DEBUG|INFO|WARN|CRITICAL|ERROR]
           Will present logs at this level of higher
           [default: INFO]
--help      Show this message and exit.
```

It is non-interactive so it breaks at the slightest provocation. An example:

```
# generates a .dot diagram of the given module
$ seqtool draw a.dot seq.samples.a
```

Create a *.dot* file from *seq.samples.a* python module. The following example generates a JPEG image from *seq.samples.b* module:

```
# generates a .png and a .jpg image of the given module
$ seqtool draw a.png seq.samples.b
$ seqtool draw a.jpg seq.samples.a
```