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: 1

Document Type: Manual (MAN)

Released on: 2021-04-24

Document Classification: Public

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

Release

This document corresponds to seq[1] v2.0.0.

Authors

Name	Affiliation
Muñoz, Iván	ESO/DOE/CSE

Change Record from previous Version

Affected Section(s)	Changes / Reason / Remarks
	See CRE ET-1084
All	All sections updated, new GUI section

[1]https://gitlab.eso.org/ifw/seq

Contents:

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 Telescopen
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*. A work in progress is the *Good Practices* section that we plan to fill in with good advice.

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 to create these graphs.

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), you are free to 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*.

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. one is creating a class and methods to control .e.g. a detector, and it is going to be instantiated many times to control many detectors in parallel, the class approach is advised.

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 wont't run until all its dependencies have finished. In the graph, a node dependency is seen as an icoming 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

Let's start with a simple sequence.

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 *do_a()* and *do_b()*. Source codes is shown below.

Listing 1: Tutorial 1 (a.py)

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

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

LOGGER = logging.getLogger("A")

async def a():
    """Simply do_a"""
    await asyncio.sleep(1)
    LOGGER.info("a")
    return "A"

async def b():
    """do_b example
    """
    LOGGER.info("B")
    return "B"

def create_sequence(*args, **kw):
    """Builds my sequence"""
    #print("logger setup")
```

(continues on next page)

(continued from previous page)

```
    #LOGGER.debug("hola poh")
    return Sequence.create(a, b, **kw)
```

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

There 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 *automagically* 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
```

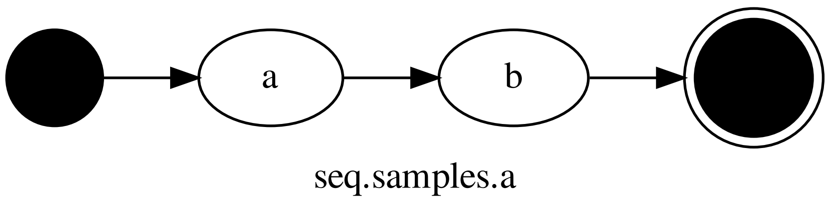


Fig. 2.1: simple sequence 01

Notice that in the *create_sequence()* function only two nodes were specified, however the *simple sequence 01* figure shows 4 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`.

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

```python
"""
Parallel nodes example.
"""
import asyncio
import random
import time
from seq.lib.nodes import Parallel, ActionInThread

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

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

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

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

Which is represented graphically as follows.

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).

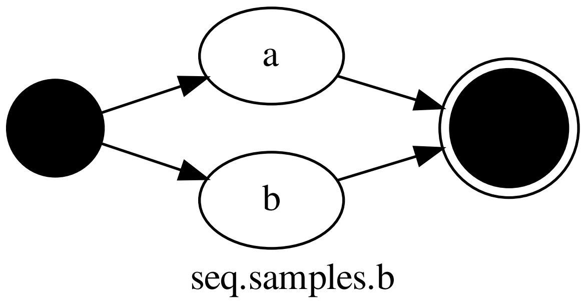


Fig. 2.2: Parallel Sequence

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

This is achieved with the `ActionInThread` node. In the example the *b()* 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

```
"""
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
```

Which is represented graphically as follows.

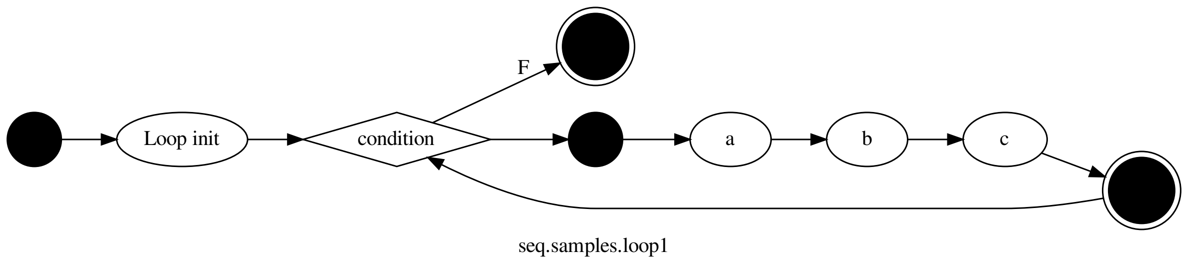


Fig. 2.3: Loop example

As the code in *Loop example* shows, the Loop's constructor takes a variable number of arguments as the Loop's body, the part that is repeated, *do_a(), do_b() and do_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 *my_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 *my_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 "Tut_01" and "Tut_02" to create a new sequence that executes them in *Parallel* and just for the kicks adds an 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()*. But how do you know which one to call? You don't have to know. You let the OB object do it for you as:

```
from seqlib.ob import OB
from seq.samples.a

mynode = OB.create_sequence(a)
```

Listing 4: Sequence embedding example

```
#!/usr/bin/env python
"""
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
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
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def create():
        aa = OB.create_sequence(a, name="A")
        bb = OB.create_sequence(b, name="B")
        s = Tpl()
        return Parallel.create(aa, bb, s.one)
```

Some points to note:

- Use *seqlib.ob.OB.create_sequence()* to select the right method to instantiate a predefined sequencer script, so it can be reused.

2.6 Conclusion

This finishes the basic tutorial. One can create any type of flow using the node types show. Mor details about node's attributes and context are given in *A Deeper Look*.

3 Sequencer GUI

The sequencer GUI allows to load and execute Python sequences and OBs (JSON format still TBD).

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

Warning: Since the server is the responsible of executing sequencer scripts, therefore standard output is swallowed by the server.

3.1 Sequencer server

The Sequencer GUI requires REDIS and the *sequencer server* to be up in order to work.

The sequencer server is started as:

```
$ seq server
```

Usage, *seq server [options]*:

```
Starts sequencer server

Options:
   --address TEXT     [HOST:]PORT
   --redis TEXT       [HOST:]PORT
   --log-level TEXT
   --help             Show this message and exit.
```

The theory is that one can have a sequencer server running on a different host than the GUI.

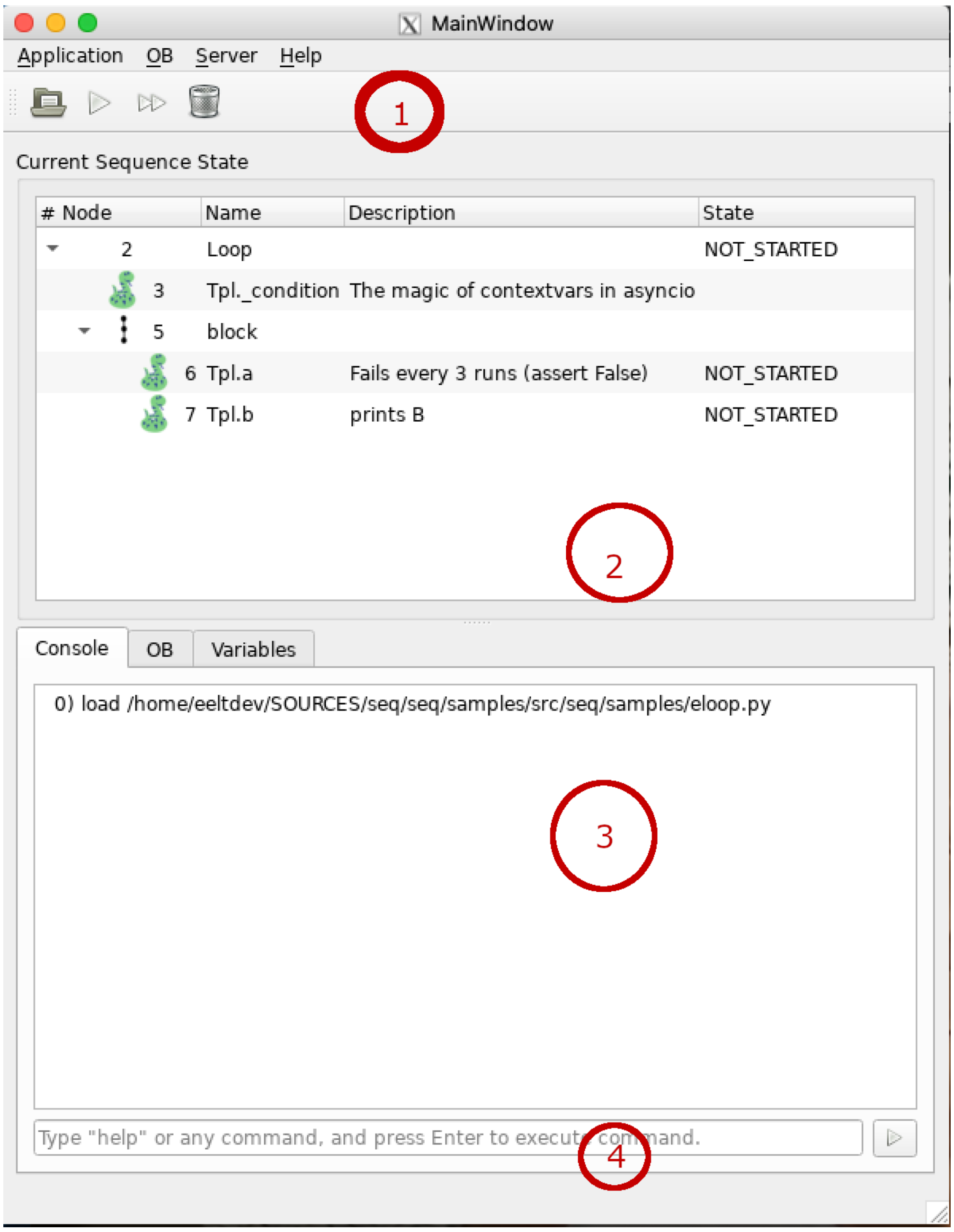
The server address is given as *[HOST:]PORT*, the same syntax is used to connect to the REDIS server.

3.2 Using the Sequencer GUI

Starting the sequencer GUI is done as follows:: $ seq gui

One can pass the *–address [HOST:]PORT* to specify the *seq server* address. The *–redis [HOST:]PORT* allows to specify the redis server address.

The GUI opens the following window.



# Node	Name	Description	State
2	Loop		NOT_STARTED
3	Tpl._condition	The magic of contextvars in asyncio	
5	block		
6	Tpl.a	Fails every 3 runs (assert False)	NOT_STARTED
7	Tpl.b	prints B	NOT_STARTED

```
0) load /home/eeltdev/SOURCES/seq/seq/samples/src/seq/samples/eloop.py
```

Type "help" or any command, and press Enter to execute command.

The GUI is divided in four parts.

1. The toolbar, displays four buttons, which functions as: *Load*, *Run*, *Continue* and *Reset*.
2. The Sequence tree. For each sequence step, displays in four columns displays: serial number, name, documentation (if any), and the state.
3. Console. The GUI communicates with a *sequencer shell* this window shows the interaction with the shell.
4. User's input. In this entry the user can type commands directed towards the sequencer shell.

The Toolbar

The GUI toolbar sports fout buttons.

Append OB Loads or Appends an OB. Multiple scripts can be loaded, until the *Run* button is pressed.

Run Executes the scripts loaded.

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

Reset Cleans the execution tree and restarts the execution engine.

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 serial number, name, doc string (from python) and its state.

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 action nodes (actual python code).

Marks Sequence nodes.

Marks Parallel nodes.

Marks Loop nodes.

The *state* column displays not only the node state but also if the node has been marked to pause (*RT.PAUSE*) or to skip its execution (*RT.SKIP*). In this context *RT* stands for *runtime flags*. 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 finish because it was puporsedly skipped (*RT.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.

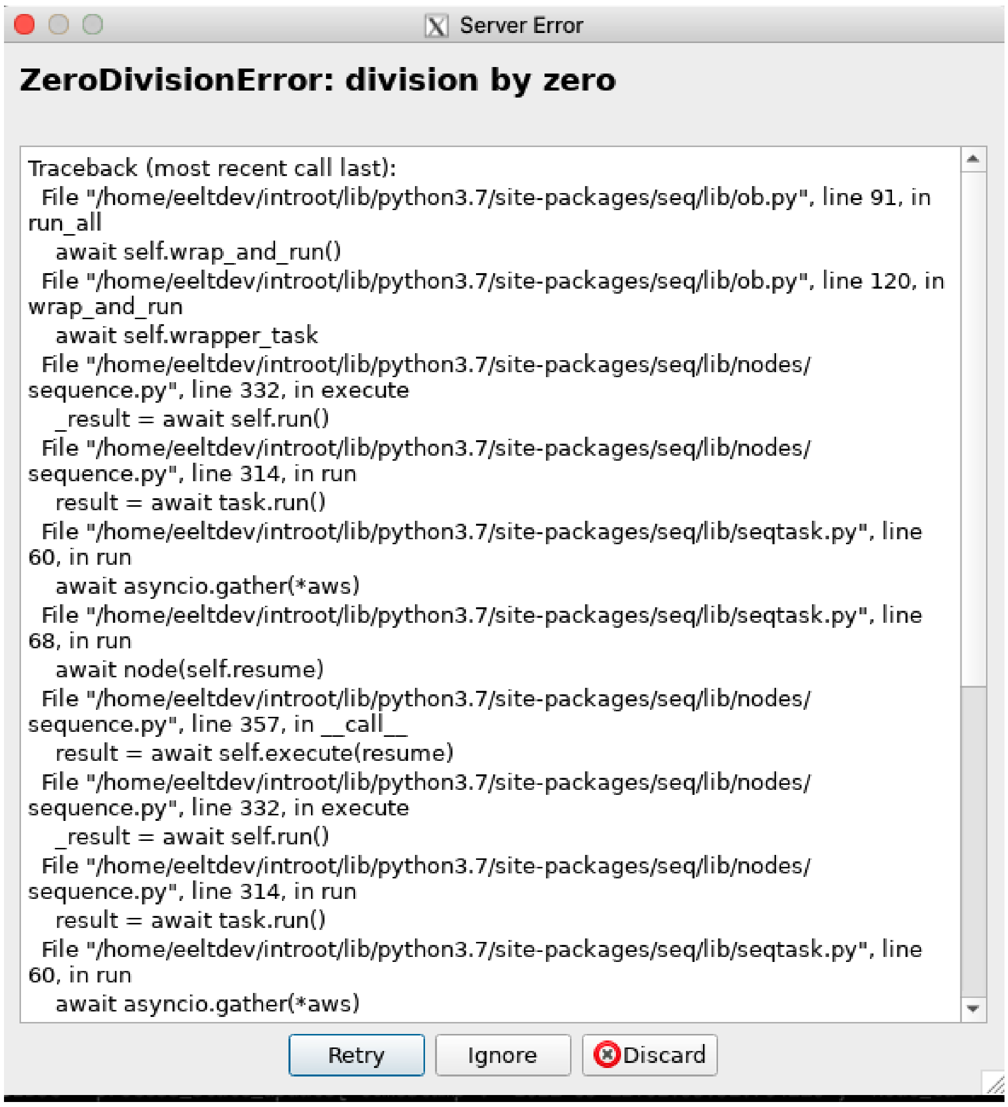
PAUSED The execution of the tree is paused in the node mark with *RT.PAUSE* flag. A paused script can be resumed by removing the *RT.PAUSE* flag from the node. This is done with the *right mouse button menu* and select *Pause/UnPause* node.

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.



ZeroDivisionError: division by zero

```
Traceback (most recent call last):
  File "/home/eeltdev/introot/lib/python3.7/site-packages/seq/lib/ob.py", line 91, in
run_all
    await self.wrap_and_run()
  File "/home/eeltdev/introot/lib/python3.7/site-packages/seq/lib/ob.py", line 120, in
wrap_and_run
    await self.wrapper_task
  File "/home/eeltdev/introot/lib/python3.7/site-packages/seq/lib/nodes/
sequence.py", line 332, in execute
    _result = await self.run()
  File "/home/eeltdev/introot/lib/python3.7/site-packages/seq/lib/nodes/
sequence.py", line 314, in run
    result = await task.run()
  File "/home/eeltdev/introot/lib/python3.7/site-packages/seq/lib/seqtask.py", line
60, in run
    await asyncio.gather(*aws)
  File "/home/eeltdev/introot/lib/python3.7/site-packages/seq/lib/seqtask.py", line
68, in run
    await node(self.resume)
  File "/home/eeltdev/introot/lib/python3.7/site-packages/seq/lib/nodes/
sequence.py", line 357, in __call__
    result = await self.execute(resume)
  File "/home/eeltdev/introot/lib/python3.7/site-packages/seq/lib/nodes/
sequence.py", line 332, in execute
    _result = await self.run()
  File "/home/eeltdev/introot/lib/python3.7/site-packages/seq/lib/nodes/
sequence.py", line 314, in run
    result = await task.run()
  File "/home/eeltdev/introot/lib/python3.7/site-packages/seq/lib/seqtask.py", line
60, in run
    await asyncio.gather(*aws)
```

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.

The Console Window

The console window just shows the output of the commands sent to the sequencer server.

The input area

Here, the user can type commands for the shell (controlled by the sequencer server) to be executed.

4 A Deeper Look

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

4.1 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 citizens 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.

4.2 Using partial functions

The use of *partial functions* allow to fix a certain number of arguments of a function and generate a **new function**, on the spot. Please see `functools.partial()`[1] for the official documentation. In any case the following example illustrates its use.

We recommend the use `seq.lib.partial()` instead of `functools.partial()`[2] since the former will keep the documentation of the wrapped function.

[1] https://docs.python.org/3/library/functools.html#functools.partial
[2] https://docs.python.org/3/library/functools.html#functools.partial

Listing 5: partial function example

```
from seq.lib import partial

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

g = f(1,2)  # creates new function g()
g(3) # Equivalent to f(1,2,3)

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.

4.3 Summary building DAGs

Some more points to note from the basic examples shown before.

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.

Instead, use the *create()* method every container node implements (Loop, Sequence, Parallel).

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 trough keyword arguments. See *Node constructor calling convention*

Listing 6: 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,
```

(continues on next page)

(continued from previous page)

```
                    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 ts *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
ALL	id	Node id (must be **unique**)
ALL	name	Node name
`Loop`	init	Initialization node
`Loop`	condition	Loop's condition node

4.4 Special Variables

Sequences uses `contextvars.ContextVar`[3] 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	current_tpl	Current Sequence name
Sequence	root	The root node
Loop	index	Loop's running index

[3] https://docs.python.org/3/library/contextvars.html#contextvars.ContextVar

4.5 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 explicitely 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 trough the *Sequence.current_tpl* 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_tpl* context variable.

Listing 7: 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_tpl.get()
  s.result = 0

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

4.6 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 trough 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_tpl* context variable as the starting node. *Lookup a node* illustrates both use cases.

Listing 8: 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_tpl.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 loosing an *Action's* result because it gets overwritten by some other node. As opposed to *Nodes have context*.

4.7 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 accesses 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 trough `Sequence.get_context()` static method. Plese see the following code excerpts *Node context example*.

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

Listing 9: 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))
```

(continues on next page)

(continued from previous page)

```
    # 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.

4.8 Node Types

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

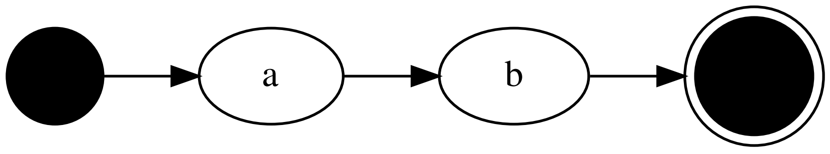
Action

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

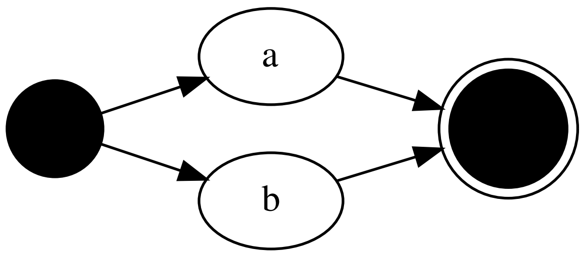
Executes nodes one after the other.



seq.samples.a

Parallel

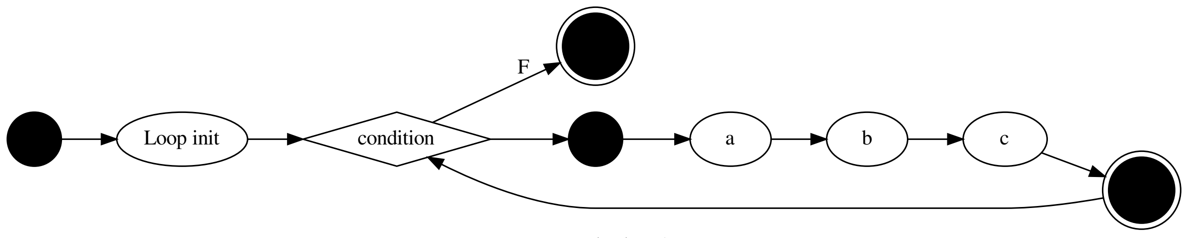
Executes nodes in parallel, finishes when all its nodes are done.



seq.samples.b

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.



seq.samples.loop1

```
"""
Implements a loop.
The condition checks Loop's index < 3.
"""
import asyncio
import logging
import random
from seq.lib.nodes import Loop
```

(continues on next page)

(continued from previous page)

```
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())
        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
```

5 Good Practices

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

5.1 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.

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 trough *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 10: 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 11: 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 12: 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)
```

6 Sequencer Command Line Tools

6.1 The *seq* meta command

The *seq* 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

The *seq 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

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.

***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.

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
```

6.2 seq run

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

```
$ seq run -h
Usage: seq run [OPTIONS] [MODULES]...

Options:
  --log-level TEXT
  --help            Show this message and exit.
```

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

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

6.3 seq draw

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

```
$ seq run -h
Usage: seq draw [OPTIONS] OUTPUT [MODULES]...

The OUTPUT argument is the generated diagram. It will
automatically generate PNG, GIF, JPG or .dot files depending on
the filename given.


Options:
  --log-level TEXT
  --help            Show this message and exit.
```

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

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

(continues on next page)

(continued from previous page)

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

Basic Usage

Command line options

The *seq 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 server. The following will load a sequence module (if found):

```
seq)>>
(seq)>> load seq.samples.a
(seq)>> (Core)> NODES: ['begin___seq_ROOT___gx13qA8w3Y', 'Sequence_YQoKAP6j2p',
↪'end___seq_ROOT___gx13qA8w3Y']
NODES: ['begin_Sequence_YQoKAP6j2p', 'a_681PO', 'b_WjBDo', 'end_Sequence_
↪YQoKAP6j2p']
```

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 ouput, 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 Inidicates 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 ti 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 ouput 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
    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 window clearly indicating the offending method and line where the exception occurrs (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 Reference

7.1 seq package

Subpackages

seqlib.nodes package

Module contents