hiro

Release 1.1.1

Ali-Akber Saifee

Jan 11, 2023

CONTENTS

1 Hiro context manager and utilities **3**

1.1 Timeline context . . . 3

1.2 run_sync and run_async . . . 5

1.2.1 Hiro context manager and utilities . . . 6

1.2.1.1 Timeline context . . . 6

1.2.1.2 run_sync and run_async . . . 8

1.2.2 API Documentation . . . 9

1.2.3 Project resources . . . 12

1.2.4 Changelog . . . 12

1.2.4.1 v1.1.1 . . . 12

1.2.4.2 v1.1.0 . . . 12

1.2.4.3 v1.0.2 . . . 12

1.2.4.4 v1.0.1 . . . 12

1.2.4.5 v1.0.0 . . . 13

1.2.4.6 v0.5.1 . . . 13

1.2.4.7 v0.5 . . . 13

1.2.4.8 v0.1.8 . . . 13

1.2.4.9 v0.1.5 . . . 13

1.2.4.10 v0.1.3 . . . 13

1.2.4.11 v0.1.2 . . . 14

1.2.4.12 v0.1.1 . . . 14

1.2.4.13 v0.0.3 . . . 14

1.2.4.14 v0.0.1 . . . 14

Index **15**

Often testing code that can be time dependent can become either fragile or slow. Hiro provides context managers and utilities to either freeze, accelerate or decelerate and jump between different points in time. Functions exposed by the standard library's `time`, `datetime` and `date` modules are patched within the contexts exposed.

CHAPTER

ONE

HIRO CONTEXT MANAGER AND UTILITIES

1.1 Timeline context

The *Timeline* context manager hijacks a few commonly used time functions to allow time manipulation within its context. Specifically `sleep()`, `time()`, `time.time_ns()`, `monotonic()`, `time.monotonic_ns()`, `time.localtime()`, `gmtime()`, `datetime.datetime.now()`, `datetime.datetime.utcnow()` and `datetime.datetime.today()` behave according the configuration of the context.

The context provides the following manipulation options:

- *rewind()*: accepts seconds as an integer or an `timedelta` instance.
- *forward()*: accepts seconds as an integer or an `timedelta` instance.
- *freeze()*: accepts a floating point time since epoch or a `datetime` or `date` instance to freeze the time at.
- *unfreeze()*: resumes time from the point it was frozen at.
- *scale()*: accepts a floating point to accelerate/decelerate time by. `> 1 = acceleration, < 1 = deceleration`
- *reset()*: resets all time alterations.

```
import hiro
from datetime import timedelta, datetime
import time

datetime.now().isoformat()
# OUT: '2013-12-01T06:55:41.706060'
with hiro.Timeline() as timeline:

    # forward by an hour
    timeline.forward(60*60)
    datetime.now().isoformat()
    # OUT: '2013-12-01T07:55:41.707383'

    # jump forward by 10 minutes
    timeline.forward(timedelta(minutes=10))
    datetime.now().isoformat()
    # OUT: '2013-12-01T08:05:41.707425'

    # jump to yesterday and freeze
    timeline.freeze(datetime.now() - timedelta(hours=24))
    datetime.now().isoformat()
```

(continues on next page)

(continued from previous page)

```
    # OUT: '2013-11-30T09:15:41'

    timeline.scale(5) # scale time by 5x
    time.sleep(5) # this will effectively only sleep for 1 second

    # since time is frozen the sleep has no effect
    datetime.now().isoformat()
    # OUT: '2013-11-30T09:15:41'

    timeline.rewind(timedelta(days=365))

    datetime.now().isoformat()
    # OUT: '2012-11-30T09:15:41'
```

To reduce the amount of statements inside the context, certain timeline setup tasks can be done via the constructor and/or by using the fluent interface.

```
import hiro
import time
from datetime import timedelta, datetime

start_point = datetime(2012,12,12,0,0,0)
my_timeline = hiro.Timeline(scale=5).forward(60*60).freeze()
with my_timeline as timeline:
    print datetime.now()
    # OUT: '2012-12-12 01:00:00.000315'
    time.sleep(5) # effectively 1 second
    # no effect as time is frozen
    datetime.now()
    # OUT: '2012-12-12 01:00:00.000315'
    timeline.unfreeze()
    # back to starting point
    datetime.now()
    # OUT: '2012-12-12 01:00:00.000317'
    time.sleep(5) # effectively 1 second
    # takes effect (+5 seconds)
    datetime.now()
    # OUT: '2012-12-12 01:00:05.003100'
```

Timeline can additionally be used as a decorator. If the decorated function expects a `timeline` argument, the *Timeline* will be passed to it.

```
import hiro
import time, datetime

@hiro.Timeline(scale=50000)
def sleeper():
    datetime.datetime.now()
    # OUT: '2013-11-30 14:27:43.409291'
    time.sleep(60*60) # effectively 72 ms
    datetime.datetime.now()
    # OUT: '2013-11-30 15:28:36.240675'
```

(continues on next page)

(continued from previous page)

```
@hiro.Timeline()
def sleeper_aware(timeline):
    datetime.datetime.now()
    # OUT: '2013-11-30 14:27:43.409291'
    timeline.forward(60*60)
    datetime.datetime.now()
    # OUT: '2013-11-30 15:28:36.240675'
```

1.2 run_sync and run_async

In order to execute certain callables within a *Timeline* context, two shortcut functions are provided.

- *hiro.run_sync()*
- *hiro.run_async()*

Both functions return a *ScaledRunner* object which provides the following methods

- *get_execution_time()*: The actual execution time of the `callable`
- *get_response()* (will either return the actual return value of `callable` or raise the exception that was thrown)

run_async() returns a derived class of *hiro.core.ScaledRunner* that additionally provides the following methods

- `is_running()`: `True/False` depending on whether the callable has completed execution
- `join()`: blocks until the `callable` completes execution

```
import hiro
import time

def _slow_function(n):
    time.sleep(n)
    if n > 10:
        raise RuntimeError()
    return n

runner = hiro.run_sync(10, _slow_function, 10)
runner.get_response()
# OUT: 10

# due to the scale factor 10 it only took 1s to execute
runner.get_execution_time()
# OUT: 1.1052658557891846

runner = hiro.run_async(10, _slow_function, 11)
runner.is_running()
# OUT: True
runner.join()
runner.get_execution_time()
# OUT: 1.1052658557891846
runner.get_response()
# OUT: Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
# ....
# OUT:   File "<input>", line 4, in _slow_function
# OUT: RuntimeError
```

1.2.1 Hiro context manager and utilities

1.2.1.1 Timeline context

The *Timeline* context manager hijacks a few commonly used time functions to allow time manipulation within its context. Specifically `sleep()`, `time()`, `time.time_ns()`, `monotonic()`, `time.monotonic_ns()`, `time.localtime()`, `gmtime()`, `datetime.datetime.now()`, `datetime.datetime.utcnow()` and `datetime.datetime.today()` behave according the configuration of the context.

The context provides the following manipulation options:

- *rewind()*: accepts seconds as an integer or an `timedelta` instance.
- *forward()*: accepts seconds as an integer or an `timedelta` instance.
- *freeze()*: accepts a floating point time since epoch or a `datetime` or `date` instance to freeze the time at.
- *unfreeze()*: resumes time from the point it was frozen at.
- *scale()*: accepts a floating point to accelerate/decelerate time by. `> 1 = acceleration, < 1 = deceleration`
- *reset()*: resets all time alterations.

```
import hiro
from datetime import timedelta, datetime
import time

datetime.now().isoformat()
# OUT: '2013-12-01T06:55:41.706060'
with hiro.Timeline() as timeline:

    # forward by an hour
    timeline.forward(60*60)
    datetime.now().isoformat()
    # OUT: '2013-12-01T07:55:41.707383'

    # jump forward by 10 minutes
    timeline.forward(timedelta(minutes=10))
    datetime.now().isoformat()
    # OUT: '2013-12-01T08:05:41.707425'

    # jump to yesterday and freeze
    timeline.freeze(datetime.now() - timedelta(hours=24))
    datetime.now().isoformat()
    # OUT: '2013-11-30T09:15:41'

    timeline.scale(5) # scale time by 5x
    time.sleep(5) # this will effectively only sleep for 1 second
```

(continues on next page)

(continued from previous page)

```
    # since time is frozen the sleep has no effect
    datetime.now().isoformat()
    # OUT: '2013-11-30T09:15:41'

    timeline.rewind(timedelta(days=365))

    datetime.now().isoformat()
    # OUT: '2012-11-30T09:15:41'
```

To reduce the amount of statements inside the context, certain timeline setup tasks can be done via the constructor and/or by using the fluent interface.

```
import hiro
import time
from datetime import timedelta, datetime

start_point = datetime(2012,12,12,0,0,0)
my_timeline = hiro.Timeline(scale=5).forward(60*60).freeze()
with my_timeline as timeline:
    print datetime.now()
    # OUT: '2012-12-12 01:00:00.000315'
    time.sleep(5) # effectively 1 second
    # no effect as time is frozen
    datetime.now()
    # OUT: '2012-12-12 01:00:00.000315'
    timeline.unfreeze()
    # back to starting point
    datetime.now()
    # OUT: '2012-12-12 01:00:00.000317'
    time.sleep(5) # effectively 1 second
    # takes effect (+5 seconds)
    datetime.now()
    # OUT: '2012-12-12 01:00:05.003100'
```

Timeline can additionally be used as a decorator. If the decorated function expects a `timeline` argument, the *Timeline* will be passed to it.

```
import hiro
import time, datetime

@hiro.Timeline(scale=50000)
def sleeper():
    datetime.datetime.now()
    # OUT: '2013-11-30 14:27:43.409291'
    time.sleep(60*60) # effectively 72 ms
    datetime.datetime.now()
    # OUT: '2013-11-30 15:28:36.240675'

@hiro.Timeline()
def sleeper_aware(timeline):
    datetime.datetime.now()
    # OUT: '2013-11-30 14:27:43.409291'
```

(continues on next page)

(continued from previous page)

```
    timeline.forward(60*60)
    datetime.datetime.now()
    # OUT: '2013-11-30 15:28:36.240675'
```

1.2.1.2 run_sync and run_async

In order to execute certain callables within a *Timeline* context, two shortcut functions are provided.

- *hiro.run_sync()*
- *hiro.run_async()*

Both functions return a *ScaledRunner* object which provides the following methods

- *get_execution_time()*: The actual execution time of the `callable`
- *get_response()* (will either return the actual return value of `callable` or raise the exception that was thrown)

run_async() returns a derived class of *hiro.core.ScaledRunner* that additionally provides the following methods

- `is_running()`: `True/False` depending on whether the callable has completed execution
- `join()`: blocks until the `callable` completes execution

```
import hiro
import time

def _slow_function(n):
    time.sleep(n)
    if n > 10:
        raise RuntimeError()
    return n

runner = hiro.run_sync(10, _slow_function, 10)
runner.get_response()
# OUT: 10

# due to the scale factor 10 it only took 1s to execute
runner.get_execution_time()
# OUT: 1.1052658557891846

runner = hiro.run_async(10, _slow_function, 11)
runner.is_running()
# OUT: True
runner.join()
runner.get_execution_time()
# OUT: 1.1052658557891846
runner.get_response()
# OUT: Traceback (most recent call last):
# ....
# OUT:   File "<input>", line 4, in _slow_function
# OUT: RuntimeError
```

1.2.2 API Documentation

class hiro.**Timeline**(*scale=1*, *start=None*)

Timeline context manager. Within this context the following builtins respect the alterations made to the timeline:

- time.time()
- time.time_ns()
- time.monotonic()
- time.monotonic_ns()
- time.sleep()
- time.localtime()
- time.gmtime()
- datetime.datetime.now()
- datetime.date.today()
- datetime.datetime.utcnow()

The class can be used either as a context manager or a decorator.

The following are all valid ways to use it.

```
with Timeline(scale=10, start=datetime.datetime(2012,12,12)):
    ....

fast_timeline = Timeline(scale=10).forward(120)

with fast_timeline as timeline:
    ....

delta = datetime.date(2015,1,1) - datetime.date.today()
future_frozen_timeline = Timeline(scale=10000).freeze().forward(delta)
with future_frozen_timeline as timeline:
    ...

@Timeline(scale=100)
def slow():
    time.sleep(120)
```

Parameters

- **scale** (*float*) – > 1 time will go faster and < 1 it will be slowed down.
- **start** – if specified starts the timeline at the given value (either a floating point representing seconds since epoch or a datetime.datetime object)

forward(*amount*)

forwards the timeline by the specified `amount`

Parameters

amount – either an integer representing seconds or a datetime.timedelta object

freeze(*target_time=None*)

freezes the timeline

Parameters

target_time – the time to freeze at as either a float representing seconds since the epoch or a `datetime.datetime` object. If not provided time will be frozen at the current time of the enclosing *Timeline*

reset()

resets the current timeline to the actual time now with a scale factor 1

rewind(*amount*)

rewinds the timeline by the specified `amount`

Parameters

amount – either an integer representing seconds or a `datetime.timedelta` object

scale(*factor*)

changes the speed at which time elapses and how long sleeps last for.

Parameters

factor (*float*) – > 1 time will go faster and < 1 it will be slowed down.

unfreeze()

if a call to *freeze()* was previously made, the timeline will be unfrozen to the point which *freeze()* was invoked.

> **Warning:** Since unfreezing will reset the timeline back to the point in when the *freeze()* was invoked - the effect of previous invocations of *forward()* and *rewind()* will be lost. This is by design so that freeze/unfreeze can be used as a checkpoint mechanism.

hiro.**run_sync**(*factor*, *func*, **args*, ***kwargs*)

Executes a callable within a *hiro.Timeline*

Parameters

- **factor** (*int*) – scale factor to use for the timeline during execution
- **func** (*function*) – the function to invoke
- **args** – the arguments to pass to the function
- **kwargs** – the keyword arguments to pass to the function

Returns

an instance of *hiro.core.ScaledRunner*

hiro.**run_threaded**(*factor*, *func*, **args*, ***kwargs*)

Executes a callable in a separate thread within a *hiro.Timeline*

Parameters

- **factor** (*int*) – scale factor to use for the timeline during execution
- **func** (*function*) – the function to invoke
- **args** – the arguments to pass to the function
- **kwargs** – the keyword arguments to pass to the function

Returns
an instance of *hiro.core.ScaledThreadedRunner*

hiro.**run_async**(*factor*, *func*, **args*, ***kwargs*)

Executes a callable in a separate thread within a *hiro.Timeline*

Parameters

- **factor** (*int*) – scale factor to use for the timeline during execution
- **func** (*function*) – the function to invoke
- **args** – the arguments to pass to the function
- **kwargs** – the keyword arguments to pass to the function

Returns
an instance of *hiro.core.ScaledAsyncRunner*

Deprecated since version 1.0.0: Use *run_threaded()*

class hiro.core.**ScaledRunner**(*factor*, *func*, **args*, ***kwargs*)

manages the execution of a callable within a *hiro.Timeline* context.

get_execution_time()

Returns
the real execution time of `func` in seconds

get_response()

Returns
the return value from `func`

Raises
Exception if the `func` raised one during execution

class hiro.core.**ScaledThreadedRunner**(**args*, ***kwargs*)

manages the threaded execution of a callable within a *hiro.Timeline* context.

get_execution_time()

Returns
the real execution time of `func` in seconds

get_response()

Returns
the return value from `func`

Raises
Exception if the `func` raised one during execution

is_running()

Rtype bool
whether the `func` is still running or not.

join()

waits for the `func` to complete execution.

hiro.core.**ScaledAsyncRunner**

alias of *ScaledThreadedRunner*

1.2.3 Project resources

Note: Hiro is tested on pythons version 3.7-3.11 and pypy 3.9

1.2.4 Changelog

1.2.4.1 v1.1.1

Release Date: 2023-01-11

Chore:

- Fix github release action

1.2.4.2 v1.1.0

Release Date: 2023-01-11

Features:

- Patch time.time_ns, time.monotonic, time.monotonic_ns, time.localtime

Bug Fix:

- Ensure time.gmtime and time.localtime work with 0 values
- Ensure Timeline can be instantiated with start=0 to reflect time=0
- Improve naming of threaded classes/functions

Deprecation:

- Deprecated *run_async* in favor of *run_threaded*

1.2.4.3 v1.0.2

Release Date: 2023-01-11

Chores:

- Update documentation theme

1.2.4.4 v1.0.1

Release Date: 2023-01-11

Compatibility:

- Update package classifiers to reflect python version support

1.2.4.5 v1.0.0

Release Date: 2023-01-10

Compatibility:

- Drop support for python < 3.7
- Update sources to not need six for compatibility

Chores:

- Standardize linting
- Add Asia/Shanghai to CI matrix

1.2.4.6 v0.5.1

Release Date: 2019-10-10

Code cleanup

1.2.4.7 v0.5

Release Date: 2017-07-26

Bug Fix:

- Account for microsecond resolution (Pull Request 3)

1.2.4.8 v0.1.8

Release Date: 2015-06-07

- Add Python 3.x support

1.2.4.9 v0.1.5

Release Date: 2014-04-03

Bug Fix:

- Imports from time weren't being patched.

1.2.4.10 v0.1.3

Release Date: 2014-04-01

Enhanced timeline decorator to pass generated timeline to decorated function

1.2.4.11 v0.1.2

Release Date: 2014-02-20

- Added blacklist for unpatchable modules
- CI improvements
- removed ScaledTimeline

1.2.4.12 v0.1.1

Release Date: 2013-12-05

Added pypy support

1.2.4.13 v0.0.3

Release Date: 2013-11-30

Allow ScaledTimeline to be used as a decorator

1.2.4.14 v0.0.1

Release Date: 2013-11-30

Initial Release

INDEX

F

forward() (*hiro.Timeline method*), 9
freeze() (*hiro.Timeline method*), 9

G

get_execution_time() (*hiro.core.ScaledRunner method*), 11
get_execution_time() (*hiro.core.ScaledThreadedRunner method*), 11
get_response() (*hiro.core.ScaledRunner method*), 11
get_response() (*hiro.core.ScaledThreadedRunner method*), 11

I

is_running() (*hiro.core.ScaledThreadedRunner method*), 11

J

join() (*hiro.core.ScaledThreadedRunner method*), 11

R

reset() (*hiro.Timeline method*), 10
rewind() (*hiro.Timeline method*), 10
run_async() (*in module hiro*), 11
run_sync() (*in module hiro*), 10
run_threaded() (*in module hiro*), 10

S

scale() (*hiro.Timeline method*), 10
ScaledAsyncRunner (*in module hiro.core*), 11
ScaledRunner (*class in hiro.core*), 11
ScaledThreadedRunner (*class in hiro.core*), 11

T

Timeline (*class in hiro*), 9

U

unfreeze() (*hiro.Timeline method*), 10