flux Documentation

Release 0.0.1

Rotem Yaari

January 21, 2016

Contents

CHAPTER 1

What is Flux?

Flux is a Python library enabling you to create virtual timelines and use them to test long processes.

The need for Flux is realized if you think about testing a code like the following:

```
def long_running_func():
    start_time = time.time()
    while not some_predicate():
        if time.time() - start_time > (60 * 60 * 24):
            LOG("This has been over a day now!!!")
            notify_long_wait()
        time.sleep(30)
```

Let's say you have a test case in place that makes sure the notification function gets triggered after one day of waiting. How do you test it?

The obvious reason is mocking the time module. However, even with mocking tools you still have to fiddle your way around the time logic to test this elegantly. You don't want to record/replay the amount of sleeps that take place, so that your test doesn't break for every change in the sleep policy. You need something that would let you express the case you're trying to achieve. Flux does just that.

Flux contains the `Timeline` class. This class lets you mock time progression, and optionally lets you register callbacks when a virtual time is reached:

```
>>> from flux import Timeline
>>> t = Timeline()
```

`Timeline.time()` behaves just like `time.time()`, only returning the virtual time. It also lets you set the

```
>>> virtual_start_time = t.time()
```

`Timeline.sleep()` behaves just like time.sleep(), sleeping the desired amount of seconds and advancing the virtual time. This seems useless, but when we shed some light on time factors its usefullness becomes more apparent.

CHAPTER 2

Time Factors

Flux allows you to control the ratio between the time in the virtual timeline and the real one. This is called a *time factor*, and is the multiplier needed on the real time to obtain the virtual time. For instance, a time factor of 2 means that for each seconds in the real world, 2 seconds will pass in the virtual timeline. Setting the time factor for a timeline object is done with the `set_time_factor function`.

Controlling the time factor is useful for many cases. For instance, you can accelerate simulated processes without changing the sleep()/time() calls in the simulating code.

A special case that is worth mentioning is the time factor of zero. It is very useful for unit tests to alleviate side effects of real time, and avoid ugly pieces of code like this:

```
>>> self.assertAlmostEquals(time(), start_time + sleep_seconds)
```

The above is usually needed because it is hard to estimate the time it takes to perform Python statements, which always adds a bit to the time deltas. Freezing your timeline will allow to test the exact effect of `sleep` and `time` calls on the timeline. This is so useful, in fact, that a shortcut exists for it – `Timeline.freeze()`.

Note: for time factor zero, calls to the `sleep` method of the timeline object are the only way to advance its current time.

CHAPTER 3

Scheduling Timed Events

`Timeline.schedule_callback()` schedules a function to be called later in the virtual timeline. It is called for you whenever someone sleeps on the virtual timeline.

```
>>> def callback():
...     print("Hello!")
>>> t.schedule_callback(100, callback)
>>> t.sleep(99) # nothing happens here
>>> t.sleep(10)
Hello!
```

So coming back to the original code, that's how we'd test it with timeline:

```
# the tested code
def long_running_func(_sleep=time.sleep, _time=time.time):
    start_time = _time()
    while not some_predicate():
        if _time() - start_time > (60 * 60 * 24):
            LOG("This has been over a day now!!!")
            notify_long_wait()
        _sleep(30)

 # testing code
 def test():
     timeline = Timeline()
     timeline.set_time_factor(0)
     timeline.schedule_callback((60 * 60 * 24) + 1, _satisfy_predicate)
     long_running_func(_sleep=timeline.sleep, _time=timeline.time)
     assert_notified_for_long_wait()
```

In some cases you might be interested in 'sleeping' until all scheduled callbacks have been triggered. This is done with `Timeline.sleep_wait_all_scheduled()`.

CHAPTER 4

Global Timeline

Flux contains a configurable global proxy, called `current_timeline`, which can be set and used by everyone. This makes writing flux-compatible code very easy:

```
try:
    from flux import current_timeline as time
except ImportError:
    import time # fallback in case flux is not installed

time.sleep(100)
```

You can replace the current global timeline at any time using `set`:

```
flux.current_timeline.set(flux.Timeline())
```

However, in most cases it should just be enough to interact with it directly, and all modules using it will automatically use the changed timeline:

```
flux.current_timeline.set_time_factor(1000000)
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search