time-travel Documentation

Release 1.1.2

Shachar Nudler

Jul 30, 2020

Contents:

time-travel is a python library that allows users to write deterministic tests for time sensitive and I/O intensive code.

time-travel supports python 2.7, 3.5, 3.6, 3.7 and pypy on both Linux and Windows.

CHAPTER 1

Quick start

1.1 Install

```
$ pip install time_travel
```

1.2 Usage

Here are two examples of how to use `time-travel`. See the full tutorial for more tips and tricks.

1.2.1 Mocking time-sensitive code

```
with TimeTravel():
    start = time.time()
    time.sleep(200)
    assert time.time() == start + 200
```

1.2.2 Mocking I/O code

```
with TimeTravel() as tt:
    sock = socket.socket()
    tt.add_future_event(time_from_now=2, sock, t.event_types.select.WRITE)

    now = time.time()
    assert select.select([], [sock], []) == ([], [sock], [])
    assert time.time() == now + 2
```

Tutorial

Why should I use time-travel?

Writing good tests can sometimes be a bit tricky, especially when you are testing code that uses a lot of I/O and has hard timing constraints.

The naïve approach for testing such code is to actually wait for the time to pass. This is bad. Horribly bad. Why?

1. Tests shouldn't take long.
2. Time is not accurate. When you wait for timeouts, there's always a threshold. If your code expects **exactly** 5 seconds to pass, there's no guarantee that `time.sleep` will wait exactly that long.

If you rely on timing in your tests - your build **will never be reliable**.

How does it work?

When loaded, the library mocks modules that access the machine's time (e.g. `time`, `datetime`) and I/O event handlers (e.g. `poll`, `select`) and replaces them with an internal event-pool implementation that lets the user choose when time moves forward and which I/O event will happen next.

The TimeTravel Context Manager

class `time_travel.`**`TimeTravel`**(*start_time=86400.0*, ***kwargs*)
Context-manager for patching time and I/O libraries.

Note: The initial time for the clock is set to *86,400* seconds since epoc. This is because Windows does not support any lower values. Sorry UNIX users!

Performance

The way the context manager works is that it changes references to patched objects in loaded modules. By default `time-travel` searches through **every loaded module** in `sys.modules`. **This takes around 2 seconds**.

Wait!! Don't leave yet!! We managed to solve this!!!

To minimize search time, `time_travel.TimeTravel` gets a keyword argument named `modules_to_patch`, which is a list of module names to search in.

For example, let's say you're testing a module named *foobar*:

```
import foobar

with TimeTravel(modules_to_patch=['foobar']) as t:
    foobar.dostuff()
```

This will reduce the replace time to the bare minimum.

Note: When the default search method is used (without the `modules_to_patch` argument) the following modules are skipped and not patched:

- `pytest`

- unittest
- mock
- threading

Moving Through Time

time.**time**()
Return the time stored in time-travel's internal clock.

time.**sleep**(*secs*)
Move time-travel's internal clock forward by *secs* seconds.

datetime.date.**today**()
Return a datetime.date object initialized to the day that time-travel's internal clock is set to.

datetime.datetime.**today**()
Return a datetime.datetime object initialized to the day that time-travel's internal clock is set to.

datetime.datetime.**now**()
Return a datetime.datetime object initialized to the time that time-travel's internal clock is set to.

datetime.datetime.**utcnow**()
Return a datetime.datetime object initialized to the time that time-travel's internal clock is set to (timezone naive).

Faking I/O Events

To mock I/O events, the user must tell time-travel which event will happen, for which file descriptor, and when. For that we have:

TimeTravel.**add_future_event**(*time_from_now*, *fd*, *event*)
Add an event to the event pool.

Parameters

- **time_from_now** – When will the event happen.
- **fd** – The descriptor (usually a socket object) that the event will happen for.
- **event** – The event that will happen (implementation specific).

For example:

```
with TimeTravel() as t:
    sock = socket.socket()
    t.add_future_event(2, sock, EVENT)
```

Note: EVENT is implementation specific for every event handler (select, poll, etc.) and will be described in the corresponding handler's documentation.

select.**select**(*rlist*, *wlist*, *xlist*, *timeout=None*)
Mimics the behaviour of select.select.

`select` has no event types, it uses positional lists in order to distinguish between *read-ready*, *write-ready* and *exception*. `TimeTravel.add_future_event()` requires an event type, so the following consts are provided:

- `TimeTravel.event_types.select.READ`
- `TimeTravel.event_types.select.WRITE`
- `TimeTravel.event_types.select.EXCEPTIONAL`

The mock returns the first event(s) that expire in the event pool and moves time forward to that point in time. For example, if the user added 2 events:

```
t.add_future_event(1, sock1, t.event_types.select.READ)
t.add_future_event(2, sock2, t.event_types.select.READ)
```

Calling `select.select([sock1, sock2], [], [])` will return an rlist containing only `sock1` and the time will move forward by 1 second.

`select.`**`poll`**`()`
Return a `MockPollObject` that behaves exactly like the real `Poll` object.

Note: This patcher is not supported on Windows.

---

class `MockPollObject`(*clock*, *event_pool*)
A mock poll object.

`MockPollObject.`**`modify`**(*fd*, *eventmask*)
Modify an already registered fd's event mask.

`MockPollObject.`**`poll`**(*timeout=None*)
Poll the set of registered file descriptors.

timeout is a value in milliseconds.

`MockPollObject.`**`register`**(*fd*, *eventmask=None*)
Register a file descriptor with the fake polling object.

`MockPollObject.`**`unregister`**(*fd*)
Remove a file descriptor tracked by the fake polling object.

The *event type* supplied to `TimeTravel.add_future_event()` is the event mask that is required by *poll.poll()* (`select.POLLIN`, `select.POLLOUT`, etc.).

Examples

Skip timeouts

Tests are deterministic and take no time with time travel. For example:

```
with TimeTravel():
    assert time.time() == 86400
    time.sleep(200)
    assert time.time() == 86600
```

```
with TimeTravel(modules_to_patch=__name__):
    assert datetime.today() == datetime.fromtimestamp(86400)
```

```
    time.sleep(250)
    assert datetime.today() == datetime.fromtimestamp(86650)
```

```
import module1
import module2

with TimeTravel(modules_to_patch=['module1', 'module2']) as time_machine:
    time_machine.set_time(100000)
    module1.very_long_method()
    module2.time_sensitive_method()
```

Patching I/O events modules

With `time-travel` you can fake future events for I/O modules:

```
with TimeTravel() as t:
    sock = socket.socket()
    t.add_future_event(2, sock, t.event_types.select.WRITE)

    now = t.clock.time
    assert select.select([], [sock], []) == ([], [sock], [])
    assert time.time() == now + 2
    assert datetime_cls.today() == datetime_cls.fromtimestamp(now + 2)
```

Or using `poll` (for supported platforms only):

```
with TimeTravel() as t:
    sock = socket.socket()
    t.add_future_event(2, sock, select.POLLIN)

    poll = select.poll()
    poll.register(sock, select.POLLIN | select.POLLOUT)

    now = t.clock.time
    assert poll.poll() == [(sock, select.POLLIN)]
    assert time.time() == now + 2
```

Implementation Details

Internally, `time-travel` has 2 main objects: a `clock`, and an `event pool`.

The Clock

The clock is an object that holds the current time (as a float), and has `listeners` that are registered to it. Whenever the time changes, the listeners's callback is called with the new time so they can react to it.

The Event Pool

The event pool keeps a set of events for different file descriptors, in different timestamps. The pool's job is to keep those events and to retrieve them for different patchers.

Writing a Patcher

Lets create a new patcher that patches the `time` module. Your patcher should inherit from `BasePatcher` and implement 2 methods:

- *get_patched_module* should return the actual module being patched by the patcher.
- *get_patch_actions* should return a list containing 3-tuples with the following information: *(object_name, the_real_object, fake_object)*

```
import time
from time_travel.patchers.basic_patcher import BasicPatcher

class MyNewPatcher(BasicPatcher):
    def get_patched_module(self):
        return time

    def get_patch_actions(self):
        return ('time.time', time.time, self._mock_time)

    def _mock_time(self):
        return 4  # Decided by a fair dice roll.
```

Adding the patcher to time-travel

`time-travel` uses entry points to add external patchers to it. For example let's imagine that our `MyNewPatcher` class is located in a file named `my_new_patcher.py`. In order to add the new patcher to `time-travel` just add the new class to the *time_travel.patchers* entry point in `setup.py`:

```
from setuptools import setup

setup(
    ...,
    entry_points={
        'time_travel.patchers' : [
            'my_new_patcher = my_new_patcher:MyNewPatcher',
        ],
    }
)
```

Event Types Hooks

If you need to hook event types to `TimeTravel.event_types` (like *`select.select()`* does) your patcher should override 2 methods:

- *get_events_namespace* should return a string that identifies the "namespace" of the event types. For example, if this returns "foo", your events will be registered under `TimeTravel.event_types.foo`.
- *get_event_types* should return an `Enum` object that contains the events.

For example:

```
from time_travel.patchers.basic_patcher import BasicPatcher

class MyNewPatcher(BasicPatcher):
    @staticmethod
```

```
    def get_events_namespace():
        return "foo"

    @staticmethod
    def get_event_types():
        return Enum("events", ['READ', 'WRITE'])
```

Index