chronometer Documentation

Release 1.0

Arthur Skowronek

January 24, 2015

Contents

Yet another simple time measurement tool for Python. The goal of this implementation is to avoid as much cruft as possible. The current version is 73 lines of actual code long, leaving out blank, doc and comment lines. Chronometer provides only functions to measure how much wall-clock time has passed between starting and stopping the timer.

Nothing more. Nothing less.

Chronometer tries to stay accurate to the actual time spent between starting and stopping the timer by utilizing a monotonic timer. According to the linux manual a monotonic timer is subject to time adjustments so it stays accurate but will never move backwards or jump. It will be adjusted gradually and always moves forward as long as the system runs.

CHAPTER 1

Examples

Easy:

```
import time
from chronometer import Chronometer

long_running_task = lambda: time.sleep(3.)

with Chronometer() as t:
    long_running_task()  # that will take a few seconds.
print('Phew, that took me {:.3f} seconds!'.format(float(t)))
```

Advanced:

```
from time import sleep
from chronometer import Chronometer

counter = 0
def long_running_task_that_can_fail():
    global counter
    counter += 1
    sleep(2.)
    return counter > 3

with Chronometer() as t:
    while not long_running_task_that_can_fail():
        print('Failed after {:.3f} seconds!'.format(t.reset()))
print('Success after {:.3f} seconds!'.format(float(t)))
```

Ridiculous:

```
import asyncio
from chronometer import Chronometer


class PingEchoServerProtocol(asyncio.StreamReaderProtocol):

    def __init__(self):
        super().__init__(asyncio.StreamReader(), self.client_connected)
        self.reader, self.writer = None, None
        self.latency_timer = Chronometer()

    def client_connected(self, reader, writer):
        self.reader, self.writer = reader, writer
        asyncio.async(self.ping_loop())
        asyncio.async(self.handler())

    @asyncio.coroutine
    def send(self, data):
        self.writer.write(data.encode('utf-8') + b'\n')
        yield from self.writer.drain()

    @asyncio.coroutine
    def ping_loop(self):
        yield from asyncio.sleep(5.)
        while True:
            if self.latency_timer.stopped:
                self.latency_timer.start()
                yield from self.send('PING (send me PONG!)')

            sleep_duration = max(2., 10. - self.latency_timer.elapsed)
            yield from asyncio.sleep(sleep_duration)

    @asyncio.coroutine
    def handler(self):
        while True:
            data = (yield from self.reader.readline())
            if data[:4] == b'PONG' and self.latency_timer.started:
                yield from self.send(('Latency: {:.3f}s'
                                      .format(self.latency_timer.stop())))

l = asyncio.get_event_loop()

@asyncio.coroutine
def startup():
    s = (yield from l.create_server(lambda: PingEchoServerProtocol(),
                                    host='localhost', port=2727))
    print('Now telnet to localhost 2727')
    yield from s.wait_closed()

l.run_until_complete(startup())
```

CHAPTER 2

API Reference

Just go read the source. Seriously. It's not that hard. If you still insist on having a documentation, here it is:

2.1 API

class `chronometer.`**`Chronometer`**`(`*timer=monotonic*`)`

Simple timer meant to be used for measuring how much time has been spent in a certain code region.

`start``()`

Starts the timer.

Returns Returns the timer itself.

Return type Chronometer

Raises TimerAlreadyStartedError If the timer is already running.

`stop``()`

Stops the timer.

Returns Time passed since the timer has been started in seconds.

Return type float

Raises TimerAlreadyStoppedError If the timer is already stopped.

`reset``()`

Resets the timer.

Returns Elapsed time before the timer was reset.

Return type float

`elapsed`

Returns time passed in seconds.

Returns Time passed since the timer has been started in seconds.

Return type float

`stopped`

Returns if the timer is stopped or not.

Returns *True* if the timer is stopped and *False* otherwise.

Return type bool

started

Returns if the timer is running or not.

Returns *True* if the timer is running and *False* otherwise.

Return type bool

class `chronometer.`**`RelaxedStartChronometer`**(*timer=monotonic*)

Relaxed version which won't raise an exception on double starting the timer.

start()

Starts the timer or just returns if the timer is already running.

Returns Returns the timer itself.

Return type RelaxedStartChronometer

class `chronometer.`**`RelaxedStopChronometer`**(*timer=monotonic*)

Relaxed version which won't raise an exception on double stopping the timer.

stop()

Stops the timer or just returns if the timer is already stopped.

Returns Time passed since the timer has been started in seconds.

Return type float

class `chronometer.`**`RelaxedChronometer`**(*timer=monotonic*)

Ultra relaxed version which won't throw any exceptions on its own.

exception `chronometer.`**`ChronoRuntimeError`**

Base exceptions for errors which happened inside Chronometer.

exception `chronometer.`**`ChronoAlreadyStoppedError`**

Raised when trying to stop a stopped timer.

exception `chronometer.`**`ChronoAlreadyStartedError`**

raised when trying to start a started timer.

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

Index