SmartTimers Documentation

Release 1.3.0

Eduardo Ponce, The University of Tennessee, Knoxville, TN

Dec 29, 2018

Contents

CHAPTER 1

SmartTimers

SmartTimers is a collection of libraries for measuring runtime of running processes using a simple and flexible API. Time can be measured in sequential and nested code blocks.

A SmartTimer allows recording elapsed time in an arbitrary number of code blocks. Specified points in the code are marked as either the beginning of a block to measure, `tic`, or as the end of a measured block, `toc`. Times are managed internally and ordered based on `tic` calls. Times can be queried, operated on, and written to file.

The following schemes are supported for timing code blocks

- Series: `tic('A'),toc(),...,tic('B'),toc()`
- Cascade: `tic('A'),toc(),toc(),...`
- Nested: `tic('A'),tic('B'),...,toc(),toc()`
- Label-paired: `tic('A'),tic('B'),...,toc('A'),toc('B')`
- Mixed: arbitrary combinations of schemes

```
from smarttimers import SmartTimer

# Create a timer instance named 'Example'
t = SmartTimer("Example")

# Print clock details
t.tic("info")
t.print_info()
t.toc()

# Measure iterations in a loop
t.tic("loop")
for i in range(10):
    t.tic("iter " + str(i))
    sum(range(1000000))
    t.toc()
t.toc()
```

(continues on next page)

(continued from previous page)

```
t.tic("sleep")
t.sleep(2)
t.toc()

# Write times to file 'Example.txt'
t.to_file()

print(t["info"])
print(t.walltime())
```

```
# Print times measured in different ways
>>> print(t)
 label,  seconds,  minutes, rel_percent, cumul_sec, cumul_min,cumul_percent
  info, 0.000270, 0.000004,       0.0001,  0.000270,  0.000004,       0.0001
  loop, 0.153422, 0.002557,       0.0664,  0.153692,  0.002562,       0.0666
iter 0, 0.022840, 0.000381,       0.0099,  0.176531,  0.002942,       0.0765
iter 1, 0.023248, 0.000387,       0.0101,  0.199780,  0.003330,       0.0865
iter 2, 0.017198, 0.000287,       0.0074,  0.216977,  0.003616,       0.0940
iter 3, 0.012921, 0.000215,       0.0056,  0.229898,  0.003832,       0.0996
iter 4, 0.012754, 0.000213,       0.0055,  0.242652,  0.004044,       0.1051
iter 5, 0.012867, 0.000214,       0.0056,  0.255519,  0.004259,       0.1107
iter 6, 0.012843, 0.000214,       0.0056,  0.268361,  0.004473,       0.1162
iter 7, 0.012789, 0.000213,       0.0055,  0.281150,  0.004686,       0.1218
iter 8, 0.012818, 0.000214,       0.0056,  0.293969,  0.004899,       0.1273
iter 9, 0.012856, 0.000214,       0.0056,  0.306825,  0.005114,       0.1329
 sleep, 2.002152, 0.033369,       0.8671,  2.308977,  0.038483,       1.0000

# Print stats only for labels with keyword 'iter'
>>> print(t.stats("iter"))
namespace(avg=(0.015313280202099122, 0.00025522133670165205),
max=(0.023248409008374438, 0.0003874734834729073),
min=(0.012753532995702699, 0.00021255888326171163),
total=(0.15313280202099122, 0.0025522133670165203))
```

CHAPTER 2

MODULES

2.1 SmartTimer

class smarttimers.smarttimer.**SmartTimer**(*name=''*, ***kwargs*)

Timer container to perform time measurements in code blocks.

Parameters

- **name** (*str, optional*) – Name of container. Default is *smarttimer*.
- **kwargs** (*dict, optional*) – Map of options to configure the internal *Timer*. Default is *Timer* defaults.

A *SmartTimer* allows recording elapsed time in an arbitrary number of code blocks. Specified points in the code are marked as either the beginning of a block to measure, *tic()*, or as the end of a measured block, *toc()*. Times are managed internally and ordered based on *tic()* calls. Times can be queried, operated on, and written to file.

The following schemes are supported for timing code blocks

- Consecutive: `tic('A'),toc(),...,tic('B'),toc()`
- Cascade: `tic('A'),toc(),toc(),...`
- Nested: `tic('A'),tic('B'),...,toc(),toc()`
- Label-paired: `tic('A'),tic('B'),...,toc('A'),toc('B')`
- Mixed: arbitrary combinations of schemes

name
str – Name of container. May be used for filename in `write_to_file()`.

labels
list, str – Label identifiers of completed timed code blocks.

active_labels
list, str – Label identifiers of active code blocks.

seconds
list, float – Elapsed time for completed code blocks.

minutes
list, float – Elapsed time for completed code blocks.

times
dict – Map of times elapsed for completed blocks. Keys are the labels used when invoking `tic()`.

__getitem__(**keys*)
Query time(s) of completed code blocks.

Parameters **keys** (*str, slice, integer*) – Key(s) to select times. If string, then consider it as a label used in `tic()`. If integer or slice, then consider it as an index (based on `tic()` ordering). Key types can be mixed.

Returns If key did not match a completed *Timer* label. list, float: Time in seconds of completed code blocks.

Return type None

tic(*label=''*)
Start measuring time.

Measure time at the latest moment possible to minimize noise from internal operations.

Parameters **label** (*str*) – Label identifier for current code block.

toc(*label=None*)
Stop measuring time at end of code block.

Note: In cascade regions, that is, multiple toc() calls, O(ms) noise will be introduced. In a future release, there is the possibility of correcting this noise, but even the correction is noise itself.

Parameters **label** (*str*) – Label identifier for current code block.

Returns Measured time in seconds.

Return type float

Raises *TimerError, KeyError* – If there is not a matching `tic()`.

walltime()
Compute elapsed time in seconds between first `tic()` and most recent `toc()`.

`walltime()` >= sum(`seconds`)

print_info()
Pretty print information of registered clock.

remove(**keys*)
Remove time(s) of completed code blocks.

Parameters **keys** (*str, slice, integer*) – Key(s) to select times. If string, then consider it as a label used in `tic()`. If integer or slice, then consider it as an index (based on `tic()` ordering). Key types can be mixed.

clear()
Empty internal storage.

reset()
Restore `name`, reset clock to default value, and empty internal storage.

dump_times (*filename=None*, *mode='w'*)
Write timing results to a file.

If *filename* is provided, then it will be used as the filename. Otherwise `name` is used if non-empty, else the default filename is used. The extension *.times* is appended only if filename does not already has an extension. Using *mode* the file can be overwritten or appended with timing data.

Parameters

- **filename** (*str, optional*) – Name of file.
- **mode** (*str, optional*) – Mode flag passed to open. Default is *w*.

stats (*label=None*)
Compute total, min, max, and average stats for timings.

Note:

- An alphanumeric label is used as a word-bounded regular expression.
- A non-alphanumeric label is compared literally.
- If *label* is 'None' then all completed timings are used.

Parameters **label** (*str, iterable, optional*) – String/regex used to match timer labels to select.

Returns Namespace with stats in seconds/minutes.

Return type types.SimpleNamespace

sleep (*seconds*)
Sleep for given seconds.

to_array ()
Return timing data as a list or numpy array (no labels).

Data is arranged as a transposed view of `__str__()` and `to_file()` formats.

Returns Timing data.

Return type numpy.ndarray, list

pic (*subcalls=True*, *builtins=True*)
Start profiling.

See profile

poc ()
Stop profiling.

print_profile (*sort='time'*)
Print profiling statistics.

dump_profile (*filename=None*, *mode='w'*)
Write profiling results to a file.

If *filename* is provided, then it will be used as the filename. Otherwise `name` is used if non-empty, else the default filename is used. The extension *.prof* is appended only if filename does not already has an extension. Using *mode* the file can be overwritten or appended with timing data.

Parameters

- **filename** (*str, optional*) – Name of file.
- **mode** (*str, optional*) – Mode flag passed to open. Default is *w*.

2.2 Timer

class smarttimers.timer.**Timer**(*label=''*, ***kwargs*)

Read current time from a clock/counter.

Parameters

- **label** (*str, optional*) – Label identifier. Default is empty string.
- **kwargs** (*dict, optional*) – Map of options. Valid options are *seconds*, *clock_name*, and timer.
- **seconds** (*float, optional*) – Time measured in fractional seconds. Default is 0.0.
- **clock_name** (*str, optional*) – Clock name used to select a time measurement function. Default is *DEFAULT_CLOCK_NAME*.
- **timer** (*Timer, optional*) – Reference instance to use as initialization.

A *Timer* allows recording the current time measured by a registered timing function. Time is recorded in fractional seconds and fractional minutes. *Timer* supports addition, difference, and logical operators. *Timer* uses a simple and extensible API which allows registering new timing functions. A timing function is compliant if it returns a time measured in fractional seconds. The function can contain arbitrary positional and/or keyword arguments or no arguments.

Listing 1: Timer API examples.

```
from smarttimers import Timer

# Find the current time function of a Timer
t1 = Timer('Timer1')
print(Timer.CLOCKS[t1.clock_name])
# or
Timer.print_clocks()
print(t1.clock_name)

# Change current time function
t1.clock_name = 'process_time'

# Record a time measurement
t1.time()
print(t1)

# Create another Timer compatible with 'Timer1'
t2 = Timer('Timer2', clock_name='process_time')
t2.print_info()
t2.time()
print(t2)

# Sum Timers
t3 = Timer.sum(t1, t2)
# or
t3 = t1 + t2
print(t3)
```

(continues on next page)

(continued from previous page)

```
28
29 # Find difference between Timers
30 t4 = Timer.diff(t1, t2)
31 # or
32 t4 = t2 - t1
33 print(t4)
34
35 # Compare Timers
36 print(t1 == t2)   # False
37 print(t2 > t1)    # True
38 print(t4 <= t3)   # True
```

Available time measurement functions in *CLOCKS*:

- 'perf_counter' -> time.perf_counter()
- 'process_time' -> time.process_time()
- 'clock' -> time.clock() (deprecated)
- 'monotonic' -> time.monotonic()
- 'time' -> time.time() (deprecated)

Listing 2: Registering a new time measurement function.

```
def custom_time_function(*args, **kwargs):
    # Measure time
    time_in_some_unit = ...

    # Convert time to fractional seconds
    time_seconds = time_in_some_unit ...
    return time_seconds

# Register custom_time_function() as 'custom_time'
Timer.register_clock('custom_time', custom_time_function)
# or
Timer.CLOCKS['custom_time'] = custom_time_function
```

Note:

- New timing functions need to have a compliant interface. If a user wants to register a non-compliant timing function, a compliant wrapper function can be used. The available timing functions are built-ins from the standard time library.
- Only Timers with compatible clocks support arithmetic and logical operators. Otherwise a `TimerCompatibilityError` exception occurs.

DEFAULT_CLOCK_NAME
str – Default clock name, used when *clock_name* is empty string.

Raises `TypeError` – If not a string.

CLOCKS
TimerDict, str -> callable – Map between clock name and time measurement functions.

Raises

- `TypeError` – If not assigned with dictionary.
- `KeyError` – If key is not a string.
- `ValueError` – If assigned item is not callable.

label
str – Label identifier.

Raises `TypeError` – If not a string.

seconds
float, read-only – Time measured in fractional seconds.

Set internally either during initialization or when recording time.

Raises

- `TypeError` – If not numeric.
- `ValueError` – If negative number.

minutes
float, read-only – Time measured in minutes.

clock_name
str – Clock name used to select a time measurement function.

Indexes the *CLOCKS* map to select a time function. If set to the empty string then *DEFAULT_CLOCK_NAME* is used. An instance is reset when set to a new and incompatible clock name.

Raises `TypeError` – If not a string.

__str__()
String representation.

Returns

Delimited string (*seconds*, *minutes*, *label*)

Return type str

time(**args*, ***kwargs*)
Record time using timing function configured via *clock_name*.

Parameters

- **args** (*tuple, optional*) – Positional arguments for time function.
- **kwargs** (*dict, optional*) – Keyword arguments for time function.

Returns Time measured in fractional seconds.

Return type float

clear()
Set time values to zero.

reset()
Clear, empty *label*, and reset clock to default value.

get_info()
Return clock information.

Returns Namespace with clock info.

Return type types.SimpleNamespace

print_info()
Pretty print clock information.

is_compatible(*timer*)
Return truth of compatibility between a *Timer* pair.

For a *clock_name* that can be queried with time.get_clock_info, compatibility requires that all attributes are identical. Otherwise the timing functions have to be the same.

Parameters **timer** (Timer) – Second instance.

Returns True if compatible, else False.

Return type bool

sleep(*seconds*)
Sleep for given seconds.

classmethod register_clock(*clock_name*, *clock_func*)
Registers a time function to *CLOCKS* map.

Parameters

- **clock_name** (*str*) – Clock name.
- **clock_func** (*callable*) – Reference to a time measurement function.

classmethod unregister_clock(*clock_name*)
Remove a registered clock from *CLOCKS* map.

Parameters **clock_name** (*str*) – Clock name.

classmethod print_clocks()
Pretty print information of registered clocks.

class smarttimers.timer.**TimerDict**(*tdict=None*)
Map between label identifier and callable object.

Parameters **tdict** (*dict, optional*) – Mapping between strings and timing functions.

Raises

- KeyError – If key is not a string or does not exists.
- ValueError – If value is not a callable object.

2.3 Decorators

SmartTimer decorators

Functions: *time()*

smarttimers.decorators.**time**(*func=None*, *, *timer=None*)
Measure runtime for functions/methods.

Parameters **timer** (SmartTimer, *optional*) – Instance to use to measure time. If None, then global SmartTimer instance, *_timer*, is used.

2.4 Exceptions

class smarttimers.exceptions.**TimerCompatibilityError**(*msg='incompatible clocks'*)
Exception for incompatible `Timer` instances.

class smarttimers.exceptions.**TimerError**
Base exception for `Timer`.

CHAPTER 3

HISTORY

3.1 v1.3.0

Date 2018-12-3

Features:

- SmartTimer: new stats() to compute total, min, max, and avg statistics.
- SmartTimer: label support for toc() in cascade scheme.
- SmartTimer: new relative and cumulative time data.
- SmartTimer: new to_array() to transform timing data to list/numpy array (numpy is optional).
- Timer, SmartTimer: added sleep().
- README: minor update to examples.
- Tests: updated some cases to support SmartTimer changes.
- Exceptions: error messages are given explicitly when raised.

3.2 v1.0.0

Date 2018-11-29

Features:

- SmartTimer: complete test cases.
- SmartTimer: consistent API behavior.
- Added basic documentation and example in README.
- Update configuration files.

3.3 v0.9.5

Date 2018-11-28

Features:

- Changed package name to smarttimers.
- SmartTimer: complete documentation.
- SmartTimer: added example.
- SmartTimer: improved exception handling.

Bug fixes:

- SmartTimer: walltime is calculated from first tic until most recent toc.
- SmartTimer: name attribute cannot be set to empty string.

3.4 v0.9.0

Date 2018-11-26

Features:

- SmartTimer: time lists maintain ordering relative to tic() calls.
- SmartTimer: support for consecutive, nested, cascade, interleaved, and key-paired code blocks.
- Improved error handling and exceptions raised.

Bug fixes:

- SmartTimer: changed times() to use new labels() output format.

3.5 v0.8.5

Date 2018-11-25

Features

- SmartTimer: implemented container for tic-toc support.
- Improved module dependencies for setup, tests, and docs.
- Code style is compliant with flake8.

3.6 v0.6.0

Date 2018-11-22

Features

- Timer: time values are read-only.
- Timer: new clear, reset, (un)register_clock methods.
- Timer: new custom exception classes and improved exception handling.

- Timer: rename sum, print_info, and print_clocks methods.
- Timer: complete documentation.
- Timer: complete test cases.
- Integrated tox, Travis CI, code coverage, and Read the Docs.
- Cleaned code adhering to flake8.

3.7 v0.4.0

Date 2018-11-19

Features

- Timer: new print/get_clock_info methods.
- Timer: improved compatibility checks.
- Timer: new TimerCompatibilityError exception.
- Timer: documentation revision.

3.8 v0.3.0

Date 2018-11-18

Features

- Timer: include an example script.
- Timer: new MetaTimerProperty and TimerDict for handling class variable properties.
- Timer: improved exception coverage.
- Timer: improved methods for checking compatibility and print clock info.
- Timer: time values are cleared automatically when clock name is changed.

3.9 v0.2.2

Date 2018-11-18

Features

- Timer: new methods to print clock details using time.get_clock_info.
- Timer: updated docstrings and provide usage examples.

3.10 v0.2.0

Date 2018-11-17

Features

- Timer: new methods to support sum, difference, and comparison operators.

- Timer: new methods to check compatibility between Timers.

3.11 v0.1.0

Date 2018-11-15

Features

- Created Timer class with timing functions from standard module time.
- Ubuntu 16.04 (Linux 4.15.0-38) support.

CHAPTER 4

CONTRIBUTING

All types of contributions are welcomed. A guide for developers will be made available soon.

Meanwhile:

- Clone repository
- Make improvements
- Fix bugs
- Make pull requests

CHAPTER 5

CREDITS

Many thanks to all people who have provided useful feedback and recommendations during the development of SmartTimer. These credits use a convention based on Linux CREDITS where the fields are: name (N), email (E), web-address (W), country (C), description (D), issues (I).

5.1 Author

```
N: Eduardo Ponce
E: edponce2010@gmail.com
W: https://github.com/edponce
C: TN, USA
D: Lead developer
```

CHAPTER 6

LICENSE

CHAPTER 7

Indices and tables

- genindex
- modindex

Python Module Index

S

Index

Symbols

A

C

D

G

I

L

M

N

P

R

S

T

U

W