gtimer Documentation

Release 1.0.0-beta.5

Adam Stooke

October 05, 2016

Contents

G-Timer is a Python timing tool intended for use cases ranging from quick, one-time measurements to permanent integration for recording project performance. The main features include:

- Flexible levels of detail: lines, functions, programs, or any combination
- Automatic organization of timing data
- Easy deployment and adjustment of measurements
- Convenient output to human-readable format or spreadsheet

CHAPTER 1

Why G-Timer?

Consider a simple in-place timing measurement:

```
t0 = timer()
some_statement
some_function()
t1 = timer()
t_elapsed = t1 - t0
print "Elapsed time: ", t_elapsed
```

It was easy, and the timing information was useful, so the program grows and so does the interest in timing detail:

```
def some_function():
    t0 = timer()
    some_statement
    some_method()
    t1 = timer()
    another_function()
    t2 = timer()
    return t2 - t1, t1 - t0

t0 = timer()
another_statement
t_some_1, t_some_2 = some_function()
t1 = timer()
another_method()
t2 = timer()
print "Total time: ", t2 - t0
print "some_method time: ", t_some_1"
print "another_function time: ", t_some_2
print "some_function time: ", t1 - t0
print "another_method time: ", t2 - t1
```

That grew cumbersome quickly! Function signatures are polluted, a mental model of timing relationships is now necessary, and adaptation to future code development will require time-consuming effort. All of these side-effects are eliminated with G-Timer:

```
import gtimer as gt
import time

@gt.wrap
def some_function():
    time.sleep(1)
    gt.stamp('some_method')
```

```
    time.sleep(2)
    gt.stamp('another_function')

some_function()
gt.stamp('some_function')
time.sleep(1)
gt.stamp('another_method')
print gt.report()
```

```
>> Total Time (s):      4.006
>>
>> Intervals
>> ---------
>> some_function ...... 3.005
>>   (some_function)
>>   some_method ........ 1.001
>>   another_function ... 2.002
>> another_method ..... 1.001
```

Code clutter is dramatically reduced, timing relationships are portrayed naturally, and adaptation is made easy. The timing data structure is built dynamically as the code executes, so the user can program G-Timer linearly and with minimal forethought. And G-Timer spans files–simply import it to act with the same timer anywhere in a program. Standard profiling is a powerful measurement alternative, but in comparison, G-Timer can streamline the interpretation of results, does not require a change in script call signature, and makes it easier to compare separate runs. Beyond this first example, more advanced capabilities are demonstrated in this documentation.

CHAPTER 2

Contents:

Sections 1-3 are for getting started. The remainder cover advanced topics.

2.1 Installation

Installation using PyPI and pip is standard:

```
pip install gtimer
```

To view the source code and / or install:

```
git clone https://github.com/astooke/gtimer
cd gtimer
pip install .
```

No third-party dependencies are required.

G-Timer is Python2- and Python3-compatible (tested in 2.7 and 3.5).

2.2 Introductory Examples

2.2.1 Starting Simple

```
import gtimer as gt
import time

time.sleep(0.1)
gt.stamp('first')
time.sleep(0.3)
gt.stamp('second')
print gt.report()
```

```
>> ---Begin Timer Report (root)---
>> Timer Name:          root (running)
>> Total Time (s):      0.4031
>> Stamps Sum:          0.4028
>> Self Time (Agg.):    4.792e-05
>>
>>
>> Intervals
```

```
>> ---------
>> first .............. 0.1024
>> second ............. 0.3004
>>
>> ---End Timer Report (root)---
```

The timer automatically starts on import, and each call to `stamp()` marks the end of an interval. The "Self Time" is how long was spent inside G-Timer functions, and has already been subtracted from the total. Times are always presented in units of seconds. The default timer name is "root", and it is indicated that this timer was still running (i.e. has not been stopped)–reports can be generated at any time without interfering with timing.

(Internally, all timing is performed using `default_timer()` imported from `timeit`.)

2.2.2 Subdividing

```python
import gtimer as gt
import time

# Could be in another file with gtimer import.
@gt.wrap
def func_1():
    time.sleep(0.1)
    gt.stamp('f_first')

def func_2():
    time.sleep(0.1)
    gt.stamp('f_inline')

time.sleep(0.1)
func_1()
gt.stamp('first')
func_2()
gt.stamp('second')
time.sleep(0.1)
gt.stamp('third', quick_print=True)
gt.subdivide('sub')
time.sleep(0.1)
func_1()
gt.stamp('sub_1')
time.sleep(0.1)
gt.stamp('sub_2')
gt.end_subdivision()
gt.stamp('fourth')
print gt.report()
```

```
>> (root) third: 0.1002
>>
>> ---Begin Timer Report (root)---
>> Timer Name:          root (running)
>> Total Time (s):      0.7049
>> Stamps Sum:          0.7037
>> Self Time (Agg.):    0.0004189
>>
>>
>> Intervals
>> ---------
```

```
>> first ............. 0.2023
>>   (func_1)
>>   f_first ........... 0.1002
>> f_inline .......... 0.1002
>> second ............ 5.96e-06
>> third ............. 0.1002
>> fourth ............ 0.301
>>   (sub)
>>   sub_1 ............. 0.2006
>>     (func_1)
>>     f_first ........... 0.1002
>>   sub_2 ............. 0.1002
>>
>> ---End Timer Report (root)---
```

Calls to `stamp()` always apply to the current level in the timer hierarchy. The time in the `'sub'` subdivision was accumulated entirely within the span of the interval `'fourth'` in the root timer. Subdivisions may be nested to any level, and subdivided times appear indented beneath the stamp to which they belong.

The negligible time of the interval `'second'` resulted from in-line timing of the un-decorated `func_2`.

Using the `quick_print` flag in a stamp prints the elapsed interval time immediately.

IMPORTANT: Subdivisions are managed according to their names. Two separate subdivisions of the same name, occuring at the same level and between stamps in the surrounding timer, will be counted as two iterations of the same timer and their data merged. If this is not the intended outcome, use distinct names.

2.2.3 Timer Control

```
time.sleep(0.1)
gt.start()
time.sleep(0.1)
gt.stamp('first')
gt.pause()
time.sleep(0.1)
gt.resume()
gt.stamp('second')
time.sleep(0.1)
gt.blank_stamp('third')
time.sleep(0.1)
gt.stop('fourth')
time.sleep(0.1)
print gt.report()
```

```
>> ---Begin Timer Report (root)---
>> Timer Name:          root
>> Total Time (s):      0.3006
>> Stamps Sum:          0.2004
>> Self Time (Agg.):    6.39e-05
>>
>>
>> Intervals
>> ---------
>> first ............. 0.1002
>> second ............ 5.96e-06
>> fourth ............ 0.1002
>>
>> ---End Timer Report (root)---
```

Calling `start()` ignores any previous time, the time between `pause()` and `resume()` is ignored, and so is any time after `stop()` (used here with an optional stamp name). The function `blank_stamp()` begins a new interval but discards the time data of the one it ends. There is finality in the `stop()` command–afterwords a timer level cannot be resumed, but it can be reset.

The "Stamps Sum" field indicates that the stamps data is not all-inclusive of the total time, due to the `blank_stamp()` call. This can also happen if stopping without stamping, or if no stamp is called immediately prior to enterting a timed loop.

2.2.4 Comparing Results

Use the `get_times()` function (or `save_pkl()`, `load_pkl()`) at the end of a completed timed run to retrieve the timing data. Collect results from multiple runs into a list and provide it to the `compare()` function to return a side-by-side comparison of timing data. For example, in an interactive session:

```
run example.py
gt.rename_root('run_1')
times_1 = gt.get_times()
run example.py
gt.rename_root('run_2')
times_2 = gt.get_times()
print gt.compare([times_1, times_2])

# inside example.py main:
gt.reset_root()
<body of script>
gt.stop()
```

```
>>                            run_1          run_2
>>                          -------        -------
>> Total.............          4.00           4.00
>> Stamps Sum........          4.00           4.00
>>
>> another_method ...          1.00           1.00
>> some_function ....          3.00           3.00
>>  some_method ......           1.00           1.00
>>  another_function .           2.00           2.00
>>
>>
>>                           Max        Min       Mean      StDev        Num
>>                         ------     ------     ------     ------     ------
>> Total .............      4.00       4.00       4.00       0.00          2
>>
>>
>> another_method ...       1.00       1.00       1.00       0.00          2
>> some_function ....       3.00       3.00       3.00       0.00          2
>>  some_method ......       1.00       1.00       1.00       0.00          2
>>  another_function .       2.00       2.00       2.00       0.00          2
```

2.3 Loops

2.3.1 Non-Unique Stamps

```
time.sleep(0.1)
gt.stamp('first')
for i in [1, 2, 3]:
    time.sleep(0.1)
    gt.stamp('loop', unique=False)
time.sleep(0.1)
gt.stamp('second')
print gt.report()
```

```
>> ---Begin Timer Report (root)---
>> Timer Name:          root
>> Total Time (s):      0.5031
>> Stamps Sum:          0.5026
>> Self Time (Agg.):    0.0001128
>>
>>
>> Intervals
>> ---------
>> first ............... 0.1017
>> loop ................ 0.3006
>> second .............. 0.1002
>>
>> ---End Timer Report (root)---
```

Setting the `unique` flag of a stamp to `False` allows it to accumulate time at every iteration. Use of the `unique` flag also allows times from disjoint segments of code to be assigned to the same stamp name. (In general, enforcing uniqueness helps prevent accidental mishandling of measurements–G-Timer uses the names of stamps and timers as identifiers.)

2.3.2 Timed For

```
time.sleep(0.1)
gt.stamp('first')
for i in gt.timed_for([1, 2, 3]):
    time.sleep(0.1)
    gt.stamp('loop_1')
    if i > 1:
        time.sleep(0.1)
        gt.stamp('loop_2')
time.sleep(0.1)
gt.stamp('second')
print gt.report()
```

```
>> ---Begin Timer Report (root)---
>> Timer Name:          root
>> Total Time (s):      0.7037
>> Stamps Sum:          0.703
>> Self Time (Agg.):    0.0002031
>>
>>
>> Intervals
```

```
>> ---------
>> first ............. 0.1017
>> loop_1 ............ 0.3006
>> loop_2 ............ 0.2004
>> second ............ 0.1003
>>
>>
>> Loop Iterations
>> ---------------
>>
>> Timer:       root
>>
>>                Total    Mean     Max      Min      Num
>>               ------  ------  ------  ------  ------
>> loop_1          0.30    0.10    0.10    0.10       3
>> loop_2          0.20    0.10    0.10    0.10       2
>>
>>
>> Iter.        loop_1        loop_2
>> -----       -------       -------
>> 0              0.10          0.10
>> 1              0.10          0.10
>> 2              0.10
>>
>>
>> ---End Timer Report (root)---
```

The loop in this example is termed an "anonymous" loop, since the intervals within it are recorded flat in the hierarchy of the surrounding code.

2.3.3 Timed While

```
time.sleep(0.1)
gt.stamp('first')
loop = gt.timed_loop('named_loop')
x = 0
while x < 3:
    next(loop)
    time.sleep(0.1)
    x += 1
    gt.stamp('loop')
loop.exit()
time.sleep(0.1)
gt.stamp('second')
print gt.report(include_itrs=False)
```

```
>> ---Begin Timer Report (root)---
>> Timer Name:          root
>> Total Time (s):      0.5035
>> Stamps Sum:          0.5028
>> Self Time (Agg.):    0.0001996
>>
>>
>> Intervals
>> ---------
>> first ............. 0.1016
>> named_loop ......... 0.3008
```

```
>>   (named_loop)
>>   loop ............... 0.3007
>> second ............. 0.1003
>>
>>
>> Loop Iterations
>> ---------------
>>
>> Timer:       root
>>
>>                Total    Mean     Max      Min      Num
>>               ------  ------  ------  ------  ------
>> named_loop      0.30    0.10    0.10    0.10       3
>>
>>
>> Timer:       named_loop
>> Lineage:     root (named_loop)
>>
>>                Total    Mean     Max      Min      Num
>>               ------  ------  ------  ------  ------
>> loop            0.30    0.10    0.10    0.10       3
>>
>>
>> ---End Timer Report (root)---
```

The `timed_loop()` command returns a timed loop object which can be iterated using either the built-in `next(loop)` or `loop.next()`. Place this as the first line inside the loop. At the first line past the loop, call `loop.exit()` to finish loop recording. The optional name provided to the loop is used in two places: as a stamp name in the surrounding timer and as the timer name for a subdivision that exists only within the loop. (In this case, with only one stamp inside the loop, that data is redundant.)

2.3.4 Timed Loop Details

`timed_loop()` and `timed_for()` both return objects that can be used as context managers:

```
with gt.timed_loop('named_loop') as loop:
    while x < 3:
        next(loop)
        do_unto_x()
        gt.stamp('loop')
```

When a `timed_for` loop used without context management needs to be broken, the loop's `exit()` must be called explicitly. Redundant exits do no harm. The `timed_loop` object can be used in both for and while loops.

Each timed loop must use a new instance. This means an inner loop object must be (re-)instantiated within the outer loop. Due to name-checking, anonymous inner loops are not supported–all inner timed loops must be named (plain inner loops using non-unique stamps are OK).

Registered Stamps

Registering stamps (using the `rgstr_stamps` timed loop keyword) will cause a 0 to be listed in any iteration in which the stamp was not encountered. This would change the report for `loop_2` in example 2. The option to register stamps is also available for subdivisions, in case of conditional stamps in subfunctions called repeatedly.

2.4 Advanced Stamp Settings

2.4.1 Setting Descriptions

Unique

If `True`, checks whether the stamp name has been used previously in the current level in timer hierarchy, and raises `UniqueNameError` if so. When inside a timed loop, G-Timer will raise the exception if ever the same stamp name is encountered twice in one iteration. Disjoint segments of code within a timed loop can be assigned to the same stamp name using `unique=False`, and the iteration data will still count according to loop iteration.

Keyword args: `unique` or `un`

Default: `True`

Keep Subdivisions

Decide whether to keep timing subdivisions which have occured since the previous stamp (each subdivision is permanently affixed to its parent timer at the first stamp call following closure of the subdivision). Perhaps a deeply nested subfunction call is not of interest for a particular run; this option can be used to ignore unwanted data without having to dig.

Keyword args: `keep_subdivisions` or `ks`

Default: `True` for `stamp()`, `False` for `b_stamp()` (only option active for `b_stamp()`)

Quick Print

One way to observe timing in progress; prints one line with the name and elapsed time newly assigned to the stamp (or total time at `stop()`).

Keyword args: `quick_print` or `qp`

Default: `False`

Save Iterations

Decide whether to save timing data for every iteration of each stamp, or else only the statistics (max, min, etc.). This setting is not applied to individual stamps, but to whole loops or subdivisions (named loops may be an exception). Set using the keyword arg `save_itrs` to `timed_loop()` and `timed_for()`. This keyword is also an optional argument to `wrap()` and `subdivide()`. In these cases, when a subfunction is called multiple times, it may not "know" that it is an iteration each time, but as timing data is accumulated, gtimer can still save individual iteration data according to this flag. This setting applies only to the immediate level at which it is applied, and does not propagate up or down the hierarchy.

Default: `True`

2.4.2 Control Options

Global Defaults

All of these settings can have their default set at any place in the code, affecting subsequent calls, using the `set_def_xx()` commands (e.g. `set_def_unique(True)`). This is active for wrapped functions–a wrap-

per with no `save_itrs` arg defines the behavior to query the current default setting at function entrance. The `b_stamp()` method is not subject to these settings; its default setting `keep_subdivisions=False` can be overridden individually but is otherwise hard-coded.

Long-form vs Short-form Keywords

The long-form and short-form variations of the keywords provided are equivalent. If both are present, they are OR'ed together. If neither is present, the current global default is used.

2.5 Parallel Applications

When using G-Timer in the context of parallel computing, with multiple separate python processes, each one will operate its own, independent G-Timer. Therefore it may be necessary to communicate parallel timing data to the master timer.

2.5.1 Communicating Raw Times

One parallel tool is the option to backdate in the `stamp()` function. This receives a time and applies a stamp in the current timer as if it happened at that time (the backdate time must be in the past but more recent than the latest stamp). A sub-process can return a time or a collection of times to the master process, so that the master need not synchronously monitor sub-process status. The effect is that timing data from a sub-process appears as if native in the master.

2.5.2 Communicating `Times` Objects

It is also possible to send `Times` data objects from sub-processes to the master and incorporate them into the master timer as subdivisions. This can be done using the `get_times()` function or `save_pkl()` for a serialized version. Disk storage could be utilized with `load_pkl()`.

Once the master process holds a collection of `Times` objects from completed sub-processes, they can be attached to the hierarchy using `attach_par_subdivision()`. In case timing data from only one representative worker is sufficient, `attach_subdivision()` can be used on a single `Times` object. In either case, the attached timing data will exist in a temporary state until the next `stamp()` call in the master timer, at which point the data will be permanently assigned to the master timer hierarchy, just as a regular subdivision ended during that interval is. To summarize, the proper sequence is:

1. stamp in master
2. run subprocesses
3. get times from subprocesses
4. attach times to master
5. stamp in master.

To stamp in the master during a sub-process run (between steps 2-4), it is recommended to first subdivide within the master, and end that subdivision before attaching. Otherwise, the master stamp containing the parallel subdivision will not reflect the duration of the parallel work.

The `compare()` function can be used to examine parallel subdivisions held within a single timer.

IMPORTANT: All timers from different sub-processes attached repeatedly as parallel subdivisions must be given distinct root names (within sub-process, e.g.: `rename_root_timer(worker_id)`). Timers with matching names

assigned to the same position and same parallel group name will be interpreted as coming from successive iterations of the same source and will have data incorrectly merged together, possibly in an undefined fashion. The parallel group name should be descriptive but the inidividual timer names could simply be the process number (will be converted via `str()`). When attaching only one representative in a loop, use the same timer name every time, regardless if the source sub-process changes.

In the future, it is hoped to incorporate a standalone memory-mapping solution for sharing data between processes without having to alter the signature of the parallel call and without having to reach to disk.

2.5.3 Independent Timing

Yet another option is to wait until program completion to collect the timing data from parallel workers to a central holding place. Then a side-by-side comparison can be reported using `compare()`.

2.5.4 Process Inheritance

Specifically in multiprocessing, it is possible that a spawned child process will inherit unwanted timing data from the master process. Use `reset_root()` to clear the history and instantiate a new underlying data structure. Persistent parallel workers with repeated task assignments could also `reset()` or `reset_root()` at the beginning of each task assignement, to export only new timing data at desired intervals.

2.6 Timing Data Structure

The timing data is held in a tree structure which is constructed dynamically as the program executes and timing data is collected. Each instance holds a dictionary of subdivisions and a dicitionary of parallel subdivisions. The subdivisions dictionary has stamp names as the keys (i.e. where does the subdivision belong) and lists of other times instances as the values. Similarly, the parallel subdivisions dictionary has stamp names as the keys, but sub-dictionaries as the values. Each of these sub-dictionaries has names of parallel groups as keys and lists of times instances (use distinct names!) as values. In the other direction, each times instance holds a reference to its parent (`None` for the root times only), and also its stamp position in the parent and, if it has one, the name of the parallel group it belongs to.

Within each times instance, overall timing data is held directly. The self time is always an aggregate including self times accumulated during activities of all subdivisions, and has already been subtracted from the total. Detailed timing data resides in a separate data structure that is a `Stamps` object instance. Within that, each element is a dictionary wherein the keys are stamp names.

If `get_times()` is called on a running timer, some of the relationships to subdivisions might not yet be determined (i.e. some subdivisions exist but the next level timer has not stamped). In this case those subdivisions appear under the 'UNASSIGNED' position, separate from any earlier iterations of the same subdivisions. The same can happen when there are subdivisions awaiting assignment either 1) at the moment a timed loop is entered or 2) when a timer is manually stopped.

2.7 Disabled Mode

G-Timer can be fully disabled by setting the environment variable 'GTIMER_DISABLE' to any value other than '0', before the first import of G-Timer. All functions will keep the same signature, but most will simply pass. Timed loops will still function, as bare loops. The status is recorded in the `gtimer.DISABLED` variable. To reenable, change the environment variable and reload gtimer.

2.8 Function Reference

gtimer.**start**(*backdate=None*)

Mark the start of timing, overwriting the automatic start data written on import, or the automatic start at the beginning of a subdivision.

Notes

Backdating: For subdivisions only. Backdate time must be in the past but more recent than the latest stamp in the parent timer.

Parameters **backdate** (*float, optional*) – time to use for start instead of current.

Returns *float* – The current time.

Raises

- `BackdateError` – If given backdate time is out of range or used in root timer.
- `StartError` – If the timer is not in a pristine state (if any stamps or subdivisions, must reset instead).
- `StoppedError` – If the timer is already stopped (must reset instead).
- `TypeError` – If given backdate value is not type float.

gtimer.**stamp**(*name*, *backdate=None*, *unique=None*, *keep_subdivisions=None*, *quick_print=None*, *un=None*, *ks=None*, *qp=None*)

Mark the end of a timing interval.

Notes

If keeping subdivisions, each subdivision currently awaiting assignment to a stamp (i.e. ended since the last stamp in this level) will be assigned to this one. Otherwise, all awaiting ones will be discarded after aggregating their self times into the current timer.

If both long- and short-form are present, they are OR'ed together. If neither are present, the current global default is used.

Backdating: record a stamp as if it happened at an earlier time. Backdate time must be in the past but more recent than the latest stamp. (This can be useful for parallel applications, wherein a sub- process can return times of interest to the master process.)

> **Warning:** When backdating, awaiting subdivisions will be assigned as normal, with no additional checks for validity.

Parameters

- **name** (*any*) – The identifier for this interval, processed through str()
- **backdate** (*float, optional*) – time to use for stamp instead of current
- **unique** (*bool, optional*) – enforce uniqueness
- **keep_subdivisions** (*bool, optional*) – keep awaiting subdivisions
- **quick_print** (*bool, optional*) – print elapsed interval time
- **un** (*bool, optional*) – short-form for unique

- **ks** (*bool, optional*) – short-form for keep_subdivisions
- **qp** (*bool, optional*) – short-form for quick_print

Returns *float* – The current time.

Raises

- `BackdateError` – If the given backdate time is out of range.
- `PausedError` – If the timer is paused.
- `StoppedError` – If the timer is stopped.
- `TypeError` – If the given backdate value is not type float.

`gtimer.`**`stop`**(*name=None*, *backdate=None*, *unique=None*, *keep_subdivisions=None*, *quick_print=None*, *un=None*, *ks=None*, *qp=None*)

Mark the end of timing. Optionally performs a stamp, hence accepts the same arguments.

Notes

If keeping subdivisions and not calling a stamp, any awaiting subdivisions will be assigned to a special 'UNASSIGNED' position to indicate that they are not properly accounted for in the hierarchy (these can happen at different places and may be combined inadvertently).

Backdating: For subdivisions only. Backdate time must be in the past but more recent than the latest stamp.

Parameters

- **name** (*any, optional*) – If used, passed to a call to stamp()
- **backdate** (*float, optional*) – time to use for stop instead of current
- **unique** (*bool, optional*) – see stamp()
- **keep_subdivisions** (*bool, optional*) – keep awaiting subdivisions
- **quick_print** (*bool, optional*) – boolean, print total time
- **un** (*bool, optional*) – see stamp()
- **ks** (*bool, optional*) – see stamp()
- **qp** (*bool, optional*) – see stamp()

Returns *float* – The current time.

Raises

- `BackdateError` – If given backdate is out of range, or if used in root timer.
- `PausedError` – If attempting stamp in paused timer.
- `StoppedError` – If timer already stopped.
- `TypeError` – If given backdate value is not type float.

`gtimer.`**`pause`**`()`

Pause the timer, preventing subsequent time from accumulating in the total. Renders the timer inactive, disabling other timing commands.

Returns *float* – The current time.

Raises

- `PausedError` – If timer already paused.

- `StoppedError` – If timer already stopped.

`gtimer.`**`resume`**`()`

Resume a paused timer, re-activating it. Subsequent time accumulates in the total.

Returns *float* – The current time.

Raises

- `PausedError` – If timer was not in paused state.
- `StoppedError` – If timer was already stopped.

`gtimer.`**`blank_stamp`**`(`*name=None*, *backdate=None*, *unique=None*, *keep_subdivisions=False*, *quick_print=None*, *un=None*, *ks=False*, *qp=None*`)`

Mark the beginning of a new interval, but the elapsed time of the previous interval is discarded. Intentionally the same signature as stamp().

Notes

The default for keep_subdivisions is False (does not refer to an adjustable global setting), meaning that any subdivisons awaiting would be discarded after having their self times aggregated into this timer. If this is set to True, subdivisions are put in the 'UNASSIGNED' position, indicating they are not properly accounted for in the hierarchy.

Parameters

- **`name`** (*`any, optional`*) – Inactive.
- **`backdate`** (*`any, optional`*) – Inactive.
- **`unique`** (*`any, optional`*) – Inactive.
- **`keep_subdivisions`** (*`bool, optional`*) – Keep subdivisions awaiting
- **`quick_print`** (*`any, optional`*) – Inactive.
- **`un`** (*`any, optional`*) – Inactive.
- **`ks`** (*`bool, optional`*) – see stamp().
- **`qp`** (*`any, optional`*) – Inactive.

Returns *float* – The current time.

Raises `StoppedError` – If timer is already stopped.

`gtimer.`**`reset`**`()`

Reset the timer at the current level in the hierarchy (i.e. might or might not be the root).

Notes

Erases timing data but preserves relationship to the hierarchy. If the current timer level was not previously stopped, any timing data from this timer (including subdivisions) will be discarded and not added to the next higher level in the data structure. If the current timer was previously stopped, then its data has already been pushed into the next higher level.

Returns *float* – The current time.

Raises `LoopError` – If in a timed loop.

`gtimer.`**`current_time`**`()`

Returns the current time using timeit.default_timer() (same as used throughout gtimer).

Returns *float* – the current time

gtimer.**subdivide**(*name*, *rgstr_stamps=None*, *save_itrs=True*)

Induce a new subdivision–a lower level in the timing hierarchy. Subsequent calls to methods like stamp() operate on this new level.

Notes

If rgstr_stamps is used, the collection is passed through set() for uniqueness, and the each entry is passed through str(). Any identifiers contained within are guaranteed to exist in the final dictionaries of stamp data when this timer closes. If any registered stamp was not actually encountered, zero values are populated. (Can be useful if a subdivision is called repeatedly with conditional stamps.)

The save_itrs input defaults to the current global default. If save_itrs is True, then whenever another subdivision by the same name is added to the same position in the parent timer, and the two data structures are merged, any stamps present only as individual stamps (but not as itrs) will be made into itrs, with each subsequent data dump (when a subdivision is stopped) treated as another iteration. (Consider multiple calls to a timer-wrapped subfunction within a loop.) This setting does not affect any other timers in the hierarchy.

Parameters

- **name** (*any*) – Identifer for the new timer, passed through str().
- **rgstr_stamps** (*list,tuple, optional*) – Identifiers.
- **save_itrs** (*bool, optional*) – Save individual iteration data.

Returns None

gtimer.**end_subdivision**()

End a user-induced timing subdivision, returning the previous level in the timing hierarchy as the target of timing commands such as stamp(). Includes a call to stop(); a previous call to stop() is OK.

Returns None

Raises

- GTimerError – If current subdivision was not induced by user.
- LoopError – If current timer is in a timed loop.

gtimer.**timed_loop**(*name=None*, *rgstr_stamps=None*, *save_itrs=True*, *loop_end_stamp=None*, *end_stamp_unique=True*, *keep_prev_subdivisions=True*, *keep_end_subdivisions=True*, *quick_print=False*)

Instantiate a TimedLoop object for measuring loop iteration timing data. Can be used with either for or while loops.

Example:

```
loop = timed_loop()
while x > 0:  # or for x in <iterable>:
    next(loop)  # or loop.next()
    <body of loop, with gtimer stamps>
loop.exit()
```

Notes

Can be used as a context manager around the loop, without requiring separate call to exit(). Redundant calls to exit() do no harm. Loop functionality is implemented in the next() or __next__() methods.

Each instance can only be used once, so for an inner loop, this function must be called within the outer loop.

Any awaiting subdivisions kept at entrance to a loop section will go to the 'UNASSIGNED' position to indicate that they are not properly accounted for in the hierarchy. Likewise for any awaiting subdivisions kept at the end of loop iterations without a named stamp.

Parameters

- **name** (*any, optional*) – Identifier (makes the loop a subdivision), passed through str().
- **rgstr_stamps** (*list, tuple, optional*) – Identifiers, see subdivision().
- **save_itrs** (*bool, optional*) – see subdivision().
- **loop_end_stamp** (*any, optional*) – Identifier, automatic stamp at end of every iteration.
- **end_stamp_unique** (*bool, optional*) – see stamp().
- **keep_prev_subdivisions** (*bool, optional*) – Keep awaiting subdivisions on entering loop.
- **keep_end_subdivisions** (*bool, optional*) – Keep awaiting subdivisions at end of iterations.
- **quick_print** (*bool, optional*) – Named loop only, print at end of each iteration.

Returns *TimedLoop* – Custom gtimer object for measuring loops.

gtimer.**timed_for**(*iterable*, *name=None*, *rgstr_stamps=None*, *save_itrs=True*, *loop_end_stamp=None*, *end_stamp_unique=True*, *keep_prev_subdivisions=True*, *keep_end_subdivisions=True*, *quick_print=False*)

Instantiate a TimedLoop object for measuring for loop iteration timing data. Can be used only on for loops.

Example:

```
for i in gtimer.timed_for(iterable, ..):
    <body of loop with gtimer stamps>
```

Notes

Can be used as a context manager around the loop. When breaking out of the loop, requires usage either as a context manager or with a reference to the object on which to call the exit() method after leaving the loop body. Redundant calls to exit() do no harm. Loop functionality is implemented in the __iter__() method.

Each instance can only be used once, so for an inner loop, this function must be called within the outer loop.

Any awaiting subdivisions kept at entrance to a loop section will go to the 'UNASSIGNED' position to indicate that they are not properly accounted for in the hierarchy. Likewise for any awaiting subdivisions kept at the end of loop iterations without a named stamp.

Parameters

- **iterable** – Same as provided to regular 'for' command.
- **name** (*any, optional*) – Identifier (makes the loop a subdivision), passed through str().
- **rgstr_stamps** (*list,tuple, optional*) – Identifiers, see subdivision().
- **save_itrs** (*bool, optional*) – see subdivision().
- **loop_end_stamp** (*any, optional*) – Identifier, automatic stamp at end of every iteration, passed through str().

- **end_stamp_unique** (*bool, optional*) – see stamp().
- **keep_prev_subdivisions** (*bool, optional*) – Keep awaiting subdivisions on entering loop.
- **keep_end_subdivisions** (*bool, optional*) – Keep awaiting subdivisions at end of iterations.
- **quick_print** (*bool, optional*) – Named loop only, print at end of each iteration.

Returns *TimedFor* – Custom gtimer object for measuring for loops.

gtimer.**reset_root**()

Re-instantiate the entire underlying timer data structure and restart (same as first import), discarding all previous state and data.

Warning: This is a hard reset without hazard checks–always executes when called, any time, anywhere.

Returns None

gtimer.**rename_root**(*name*)

Rename the root timer (regardless of current timing level).

Parameters **name** (*any*) – Identifier, passed through str()

Returns *str* – Implemented identifier.

gtimer.**set_save_itrs_root**(*setting*)

Adjust the root timer save_itrs setting, such as for use in multiprocessing, when a root timer may become a parallel subdivision (see subdivide()).

Parameters **setting** (*bool*) – Save individual iterations data, passed through bool()

Returns *bool* – Implemented setting value.

gtimer.**rgstr_stamps_root**(*rgstr_stamps*)

Register stamps with the root timer (see subdivision()).

Parameters **rgstr_stamps** (*list, tuple*) – Collection of identifiers, passed through set(), then each is passed through str().

Returns *list* – Implemented registered stamp collection.

gtimer.**set_def_save_itrs**(*setting*)

Set the global default (henceforth) behavior whether to save individual iteration data of new subdivisions and loops.

Parameters **setting** – Passed through bool().

Returns *bool* – Implemented setting value.

gtimer.**set_def_keep_subdivisions**(*setting*)

Set the global default (henceforth) behavior whether to keep awaiting subdivisions when stamping.

Parameters **setting** – Passed through bool().

Returns *bool* – Implemented setting value.

gtimer.**set_def_quick_print**(*setting*)

Set the global default (henceforth) behavior whether to quick print when stamping or stopping.

Parameters **setting** – Passed through bool().

Returns *bool* – Implemented setting value.

gtimer.**set_def_unique**(*setting*)
Set the global default (henceforth) behavior whether to enforce unique stamp names (recommended).

Parameters **setting** – Passed through bool().

Returns *bool* – Implemented setting value.

gtimer.**get_times**()
Produce a deepcopy of the current timing data (no risk of interference with active timing or other operaitons).

Returns *Times* – gtimer timing data structure object.

gtimer.**save_pkl**(*filename=None*, *times=None*)
Serialize and / or save a Times data object using pickle (cPickle).

Parameters

- **filename** (*None, optional*) – Filename to dump to. If not provided, returns serialized object.
- **times** (*None, optional*) – object to dump. If non provided, uses current root.

Returns *pkl* – Pickled Times data object, only if no filename provided.

Raises TypeError – If 'times' is not a Times object or a list of tuple of them.

gtimer.**load_pkl**(*filenames*)
Unpickle file contents.

Parameters **filenames** (*str*) – Can be one or a list or tuple of filenames to retrieve.

Returns *Times* – A single object, or from a collection of filenames, a list of Times objects.

Raises TypeError – If any loaded object is not a Times object.

gtimer.**attach_par_subdivision**(*par_name*, *par_times*)
Manual assignment of a collection of (stopped) Times objects as a parallel subdivision of a running timer.

Notes

An example sequence of proper usage:

1. Stamp in master process.
2. Run timed sub-processes.
3. Get timing data from sub-processes into master.
4. Attach timing data (i.e. list of Times objects) in master using this method.
5. Stamp in master process.

To stamp in the master between steps 1 and 5, it is recommended to subdivide() between steps 1 and 2, and end that subdivision before attaching, or else the master stamp will not reflect the sub-process time.

Parameters

- **par_name** (*any*) – Identifier for the collection, passed through str()
- **par_times** (*list or tuple*) – Collection of Times data objects.

Raises TypeError – If par_times not a list or tuple of Times data objects.

gtimer.**attach_subdivision**(*times*)
Manual assignment of a (stopped) times object as a subdivision of running timer. Use cases are expected to be very limited (mainly provided as a one-Times variant of attach_par_subdivision).

Notes

As with any subdivision, the interval in the receiving timer is assumed to totally subsume the time accumulated within the attached object–the total in the receiver is not adjusted!

Parameters **times** (*Times*) – Individual Times data object.

Raises TypeError – If times not a Times data object.

gtimer.**report**(*times=None*, *include_itrs=True*, *include_stats=True*, *delim_mode=False*, *format_options=None*)

Produce a formatted report of the current timing data.

Notes

When reporting a collection of parallel subdivisions, only the one with the greatest total time is reported on, and the rest are ignored (no branching). To compare parallel subdivisions use compare().

Parameters

- **times** (*Times, optional*) – Times object to report on. If not provided, uses current root timer.
- **include_itrs** (*bool, optional*) – Display invidual iteration times.
- **include_stats** (*bool, optional*) – Display iteration statistics.
- **delim_mode** (*bool, optional*) – If True, format for spreadsheet.
- **format_options** (*dict, optional*) – Formatting options, see below.

Formatting Keywords & Defaults:

Human-Readable Mode

- 'stamp_name_width': 20
- 'itr_tab_width': 2
- 'itr_num_width': 6
- 'itr_name_width': 12
- 'indent_symbol': ' ' (two spaces)
- 'parallel_symbol': '(par)'

Delimited Mode

- 'delimiter': ' ' (tab)
- 'ident_symbol': '+'
- 'parallel_symbol': '(par)'

Returns *str* – Timing data report as formatted string.

Raises TypeError – If 'times' param is used and value is not a Times object.

gtimer.**compare**(*times_list=None*, *name=None*, *include_list=True*, *include_stats=True*, *delim_mode=False*, *format_options=None*)

Produce a formatted comparison of timing datas.

Notes

If no times_list is provided, produces comparison reports on all parallel subdivisions present at the root level of the current timer. To compare parallel subdivisions at a lower level, get the times data, navigate within it to the parallel list of interest, and provide that as input here. As with report(), any further parallel subdivisions encountered have only their member with the greatest total time reported on (no branching).

Parameters

- **times_list** (*Times, optional*) – list or tuple of Times objects. If not provided, uses current root timer.
- **name** (*any, optional*) – Identifier, passed through str().
- **include_list** (*bool, optional*) – Display stamps hierarchy.
- **include_stats** (*bool, optional*) – Display stamp comparison statistics.
- **delim_mode** (*bool, optional*) – If True, format for spreadsheet.
- **format_options** (*None, optional*) – Formatting options, see below.

Formatting Keywords & Defaults:

Human-readable Mode

- 'stamp_name_width': 18
- 'list_column_width': 12
- 'list_tab_width': 2
- 'stat_column_width': 8
- 'stat_tab_width': 2
- 'indent_symbol: ' ' (one space)

Delimited Mode

- 'delimiter': ' ' (tab)
- 'ident_symbol': '+'

Returns *str* – Times data comparison as formatted string.

Raises TypeError – If any element of provided collection is not a Times object.

gtimer.**write_structure**(*times=None*)

Produce a formatted record of a times data structure.

Parameters **times** (*Times, optional*) – If not provided, uses the current root timer.

Returns *str* – Timer tree hierarchy in a formatted string.

Raises TypeError – If provided argument is not a Times object.

2.9 Change Log

2.9.1 v1.0.0.b.5

Bug fixes:

- `load_pkl` no longer broken (used to attemp to load each file twice)

2.9.2 v1.0.0.b.4

Changes:

- Python3 compatibility.
- Commented out all mmap functions (likely weren't functional yet anyway.)

2.9.3 v1.0.0.b.3

Changes:

- Added `current_time` function.
- Removed `backdate_stamp` function, and built backdating into `start`, `stamp`, and `stop`.

2.9.4 v1.0.0.b.2

Changes:

- Added `backdate_stamp` function.
- Previously named `b_stamp` function is now called `blank_stamp`.

2.9.5 v1.0.0.b.1

Changes:

- Append "(running)" to timer name when reporting on root timer that is not stopped.

Bug fixes:

- Incomplete internal reporting call structure when providing times object.

2.9.6 v1.0.0.b.0

Initial release.

CHAPTER 3

Indices and Tables

- genindex
- search

g