Measuring elapsed time with the Time module

Asked 14 years, 3 months ago Modified 2 years, 11 months ago Viewed 552k times



With the Time module in python is it possible to measure elapsed time? If so, how do I do that?

381



I need to do this so that if the cursor has been in a widget for a certain duration an event happens.

python time elapsed

47)

Share
Improve this question
Follow



N.B. that any answer using time.time() is incorrect. The simplest example is if the system time gets changed during the measurement period. — OrangeDog Dec 4, 2017 at 16:46

For your original question regarding firing an event if a cursor stays for a certain duration on a widget, docs.python.org/3/library/threading.html provides all you need, I think. Multithreading and a condition variable with timeout might be one of the solutions. Your circumstances, however, are currently unclear to answer. — Tora Mar 18, 2018 at 18:04

There is no reason anyone should be using time.time() to measure elapsed time in modern python (affected by manual changes, drift, leap seconds etc). This answer below needs to be higher, considering this question is now top result in Google for measuring elapsed time. – NPras Nov 7, 2018 at 0:42

You can measure time with the cProfile profiler as well: docs.python.org/3/library/profile.html#module-cProfile stackoverflow.com/questions/582336/... — Anton Tarasenko Dec 19, 2018 at 12:54

0 @NPras forget "modern python". It was always incorrect to
use time.time(). - OrangeDog May 18, 2020 at 21:20

10 Answers

Sorted by:

Highest score (default)





```
start_time = time.time()
# your code
elapsed_time = time.time() - start_time
```



559

You can also write simple decorator to simplify measurement of execution time of various functions:







```
import time
from functools import wraps

PROF_DATA = {}

def profile(fn):
    @wraps(fn)
    def with_profiling(*args, **kwargs):
        start_time = time.time()

    ret = fn(*args, **kwargs)
```

```
elapsed_time = time.time() - start_time
        if fn.__name__ not in PROF_DATA:
            PROF_DATA[fn.\_name\_] = [0, []]
        PROF_DATA[fn.\_name\_][0] += 1
        PROF_DATA[fn.__name__][1].append(elapsed_time)
        return ret
    return with_profiling
def print_prof_data():
    for fname, data in PROF_DATA.items():
        max\_time = max(data[1])
        avg_time = sum(data[1]) / len(data[1])
        print "Function %s called %d times. " % (fname
        print 'Execution time max: %.3f, average: %.3f
def clear_prof_data():
    global PROF_DATA
    PROF_DATA = \{\}
```

Usage:

```
@profile
def your_function(...):
    ...
```

You can profile more then one function simultaneously. Then to print measurements just call the print_prof_data():

Share Improve this answer Follow

edited Feb 2, 2013 at 19:53
user2035895
3 • 2



- You can also take a look at <u>profilehooks</u> pip install profilehooks, and its <u>homepage here</u> pjama Jan 5, 2013 at 5:08
- 19 Note that since Python 3.3, one should probably use time.monotonic() rather then time.time() when measuring timeouts or durations.

 docs.python.org/3/library/time.html#time.monotonic

 Debilski May 18, 2015 at 21:53
- Worth adding/noting here that the unit of measure for elapsed time will be seconds. Eric Kramer Jan 16, 2016 at 3:59
- @EricKramer thank you! huge pet peev of mine, explaining measurements without defining the unit of measurement. And as a .NET guy dipping his toes into Python for the first time, I automatically thought "milliseconds". – Adam Plocher Nov 30, 2016 at 1:01
- Doesn't work if (e.g.) the system clock is changed, and may not have subsecond resolution. Correct answer: stackoverflow.com/a/47637891/476716 – OrangeDog Dec 4, 2017 at 16:44



time.time() will do the job.

98

start
run
end =

```
start = time.time()
# run your code
end = time.time()
```

import time



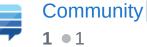
elapsed = end - start

You may want to look at <u>this</u> question, but I don't think it will be necessary.

Share Improve this answer Follow

edited May 23, 2017 at 12:26

Community Bot



answered Sep 1, 2010 at 18:28



6,303 • 7 • 43 • 56

9 Yes, time is in seconds – Eric Kramer Jan 16, 2016 at 3:58

You should change start to start_time. – Zoran Pandovski Jan 30, 2020 at 14:24

time.time() is a bad idea because the system clock can
be reset which will make you go back in time.
time.monotonic() takes care of this (monotonic = it only
goes forward). time.perf_counter() is also monotonic
but has even higher accuracy, so this is recommended for
wall-clock time. - xjcl May 31, 2020 at 11:19



For users that want better formatting,

96



import time
start_time = time.time()
your script
elapsed_time = time.time() - start_time
time.strftime("%H:%M:%S", time.gmtime(elapsed_time))



will print out, for 2 seconds:

```
'00:00:02'
```

and for 7 minutes one second:

```
'00:07:01'
```

note that the minimum time unit with gmtime is seconds. If you need microseconds consider the following:

```
import datetime
start = datetime.datetime.now()
# some code
end = datetime.datetime.now()
elapsed = end - start
print(elapsed)
# or
print(elapsed.seconds,":",elapsed.microseconds)
```

strftime documentation

Share Improve this answer Follow

edited Jan 24, 2018 at 21:42

Emma Strubell

655 • 5 • 18

answered Oct 3, 2017 at 12:04



Thank you for your answer, which inspires me. I am going to
use e = time.time() - start_time;
print("%02d:%02d:%02d" % (e // 3600, (e % 3600 //

60), (e % 60 // 1))) that yields almost same as well as covering the situation elapsing more than 24 hours. – Tora Mar 18, 2018 at 17:06

@Tora you might want to check out "{}".format() instead of %02d for future compatibility issues. – Rutger Hofste Mar 19, 2018 at 9:26

2 thank you! Now I am getting used to the new one. '{:02d}: {:02d}: {:02d}'.format(e // 3600, (e % 3600 // 60), e % 60) - Tora Apr 15, 2018 at 18:38

can you use time.monotonic() as in the other answers?

- endolith Sep 5, 2019 at 20:19

elapsed.seconds will be incorrect if the duration is greater than one day. You want elapsed.total_seconds() to be resilient – Ash Berlin-Taylor Jun 3, 2020 at 14:36



For the best measure of elapsed time (since Python 3.3), use time.perf_counter().

71







Return the value (in fractional seconds) of a performance counter, i.e. a clock with the highest available resolution to measure a short duration. It does include time elapsed during sleep and is system-wide. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

For measurements on the order of hours/days, you don't care about sub-second resolution so use

time.monotonic() instead.

Return the value (in fractional seconds) of a monotonic clock, i.e. a clock that cannot go backwards. The clock is not affected by system clock updates. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

In many implementations, these may actually be the same thing.

Before 3.3, you're stuck with time.clock().

On Unix, return the current processor time as a floating point number expressed in seconds. The precision, and in fact the very definition of the meaning of "processor time", depends on that of the C function of the same name.

On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function QueryPerformanceCounter(). The resolution is typically better than one microsecond.

Update for Python 3.7

New in Python 3.7 is <u>PEP 564</u> -- Add new time functions with nanosecond resolution.

Use of these can further eliminate rounding and floatingpoint errors, especially if you're measuring very short periods, or your application (or Windows machine) is long-running.

Resolution starts breaking down on perf_counter() after around 100 days. So for example after a year of uptime, the shortest interval (greater than 0) it can measure will be bigger than when it started.

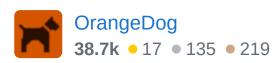
Update for Python 3.8

time.clock is now gone.

Share Improve this answer Follow

edited May 18, 2020 at 21:17

answered Dec 4, 2017 at 16:34



"In many implementations, these may actually be the same thing." True, on my Linux Mint PC, time.monotonic() and time.perf_counter() seem to return identical values. – xjcl May 31, 2020 at 11:54



In programming, there are *2 main ways to measure time*, with different results:

8



```
43
```

```
>>> print(time.process_time()); time.sleep(10); print(
0.11751394000000001

0.11764988400000001 # took 0 seconds and a bit
>>> print(time.perf_counter()); time.sleep(10); print(
3972.465770326

3982.468109075 # took 10 seconds and a bit
```

- Processor Time: This is how long this specific process spends actively being executed on the CPU.
 Sleep, waiting for a web request, or time when only other processes are executed will not contribute to this.
 - Use time.process_time()
- Wall-Clock Time: This refers to how much time has passed "on a clock hanging on the wall", i.e. outside real time.
 - Use time.perf_counter()
 - time.time() also measures wall-clock time but can be reset, so you could go back in time
 - time.monotonic() cannot be reset
 (monotonic = only goes forward) but has
 lower precision than time.perf_counter()

Share Improve this answer Follow





For a longer period.

7

```
import time
start_time = time.time()
...
e = int(time.time() - start_time)
print('{:02d}:{:02d}:{:02d}'.format(e // 3600, (e % 36))
```



would print

```
00:03:15
```

if more than 24 hours

```
25:33:57
```

That is inspired by Rutger Hofste's answer. Thank you Rutger!

```
Share Improve this answer edited Apr 15, 2018 at 18:36 Follow
```

answered Mar 18, 2018 at 17:24





You need to import time and then use time.time() method to know current time.

5







import time
start_time=time.time() #taking current time as startin
#here your code
elapsed_time=time.time()-start_time #again taking curr

Share Improve this answer Follow

edited Oct 28, 2016 at 23:12

answered Mar 10, 2016 at 23:33



Nurul Akter Towhid **3,236** • 2 • 35 • 36



Another nice way to time things is to use the **with** python structure.

5



structure.

with structure is automatically calling __enter__ and __exit__ methods which is exactly what we need to time things.



Let's create a *Timer* class.

1

```
from time import time

class Timer():
    def __init__(self, message):
        self.message = message
    def __enter__(self):
        self.start = time()
        return None # could return anything, to be us
Timer("Message") as value:
    def __exit__(self, type, value, traceback):
```

```
elapsed_time = (time() - self.start) * 1000
print(self.message.format(elapsed_time))
```

Then, one can use the Timer class like this:

```
with Timer("Elapsed time to compute some prime numbers
   primes = []
   for x in range(2, 500):
        if not any(x % p == 0 for p in primes):
            primes.append(x)
    print("Primes: {}".format(primes))
```

The result is the following:

```
Primes: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499]
```

Elapsed time to compute some prime numbers: 5.01704216003418ms

Share Improve this answer

answered Jun 25, 2019 at 8:16

T.M.

629 • 9 • 10



Vadim Shender response is great. You can also use a simpler decorator like below:

2







```
import datetime
def calc_timing(original_function):
    def new_function(*args, **kwargs):
        start = datetime.datetime.now()
        x = original_function(*args, **kwargs)
        elapsed = datetime.datetime.now()
        print("Elapsed Time = {0}".format(elapsed-star return x
        return new_function()
@calc_timing
def a_func(*variables):
    print("do something big!")
```

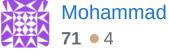
Share Improve this answer

import collections

Follow

answered Oct 17, 2018 at 4:18

Mohammad





Here is an update to Vadim Shender's clever code with tabular output:

0







import time
from functools import wraps

PROF_DATA = collections.defaultdict(list)

def profile(fn):
 @wraps(fn)
 def with_profiling(*args, **kwargs):
 start_time = time.time()
 ret = fn(*args, **kwargs)

elapsed_time = time.time() - start_time

```
PROF_DATA[fn.__name__].append(elapsed_time)
        return ret
    return with_profiling
Metrics = collections.namedtuple("Metrics", "sum_time
max_time avg_time fname")
def print_profile_data():
    results = []
    for fname, elapsed_times in PROF_DATA.items():
        num_calls = len(elapsed_times)
        min time = min(elapsed times)
        max_time = max(elapsed_times)
        sum_time = sum(elapsed_times)
        avg_time = sum_time / num calls
        metrics = Metrics(sum_time, num_calls, min_tim
fname)
        results.append(metrics)
    total_time = sum([m.sum_time for m in results])
    print("\t".join(["Percent", "Sum", "Calls", "Min",
"Function"]))
    for m in sorted(results, reverse=True):
        print("%.1f\t%.3f\t%d\t%.3f\t%.3f\t%s" %
total_time, m.sum_time, m.num_calls, m.min_time, m.max
m.fname))
    print("%.3f Total Time" % total_time)
```

Share Improve this answer edited Sep 27, 2019 at 13:43 Follow

answered Sep 27, 2019 at 13:27

