Measure Time in Python – time.time() vs time.clock()

A prerequisite before we dive into the difference of measuring time in Python is to understand various types of time in the computing world. The first type of time is called CPU or execution time, which measures how much time a CPU spent on executing a program. The second type of time is called wall-clock time, which measures the total time to execute a program in a computer. The wall-clock time is also called elapsed or running time. Compared to the CPU time, the wall-clock time is often longer because the CPU executing the measured program may also be executing other program's instructions at the same time.

Another important concept is the so-called system time, which is measured by the system clock. System time represents a computer system's notion of the passing of time. One should remember that the system clock could be modified by the operating system, thus modifying the system time.

Python's `time` module provides various time-related functions. Since most of the time functions call platform-specific C library functions with the same name, the semantics of these functions are platform-dependent.

time.time vs time.clock

Two useful functions for time measurement are `time.time` and `time.clock`. `time.time` returns the time in seconds since the epoch, i.e., the point where the time starts. For any operatin system, you can always run time.gmtime(0) to find out what epoch is on the given system. For Unix, the epoch is January 1, 1970. For Windows, the epoch is January 1, 1601. `time.time` is often used to benchmark a program on Windows. While `time.time` behaves the same on Unix and on Windows, `time.clock` has different meanings. On Unix, `time.clock` returns the current processor time expressed in seconds, i.e., the CPU time it takes to execute the current thread so far. While on Windows, it returns the wall-clock time expressed in seconds elapsed since the first call to this function, based on the Win32 function `QueryPerformanceCounter`. Another difference between `time.time` and `time.clock` is that `time.time` could return a lower-value than a previous call if the system clock has been set back between the two calls while `time.clock` always return non-decreasing values.

Here is an example of running `time.time` and `time.clock` on a Unix machine:

On a Unix-based OS

```
>>> import time
>>> print(time.time(), time.clock())
```

`time.time()` shows that the wall-clock time has passed approximately one second while `time.clock()` shows the CPU time spent on the current process is less than 1 microsecond. `time.clock()` has a much higher precision than `time.time()`.

Running the same program under Windows gives back completely different results:

On Windows

```
>>> import time
>>> print(time.time(), time.clock())
1359147763.02 4.95873078841e-06
>>> time.sleep(1)
>>> print(time.time(), time.clock())
1359147764.04 1.01088769662
```

Both `time.time()` and `time.clock()` show that the wall-clock time passed approximately one second. Unlike Unix, `time.clock()` does not return the CPU time, instead it returns the wall-clock time with a higher precision than `time.time()`.

Given the platform-dependent behavior of `time.time()` and `time.clock()`, which one should we use to measure the "exact" performance of a program? Well, it depends. If the program is expected to run in a system that almost dedicates more than enough resources to the program, i.e., a dedicated web server running a Python-based web application, then measuring the program using `time.clock()` makes sense since the web application probably will be the major program running on the server. If the program is expected to run in a system that also runs lots of other programs at the same time, then measuring the program using `time.time()` makes sense. Most often than not, we should use a wall-clock-based timer to measure a program's performance since it often reflects the productions environment.

The timeit module

Instead of dealing with the different behaviors of `time.time()` and `time.clock()` on different platforms, which is often error-prone, Python's `timeit` module provides a simple way for timing. Besides calling it directly from code, you can also call it from the command-line.

For example:

On a Unix-based OS

```
% python -m timeit -n 10000 '[v for v in range(10000)]'
10000 loops, best of 3: 365 usec per loop
% python -m timeit -n 10000 'map(lambda x: x^2, range(1000))'
10000 loops, best of 3: 145 usec per loop
```

On Windows

```
C:\Python27>python.exe -m timeit -n 10000 "[v for v in range(10000)]"
10000 loops, best of 3: 299 usec per loop
C:\Python27>python.exe -m timeit -n 10000 "map(lambda x: x^2, range(1000))"
10000 loops, best of 3: 109 usec per loop
```

In IDLE

```
>>> import timeit
>>> total_time = timeit.timeit('[v for v in range(10000)]', number=10000)
>>> print(total_time)
3.60528302192688  # total wall-clock time to execute the statement 10000 times
>>> print(total_time / 10000)
0.00036052830219268796  # average time per loop
>>> total_time = timeit.timeit('[v for v in range(10000)]', number=10000)
>>> print(total_time)
3.786295175552368  # total wall-lock time to execute the statement 10000 times
>>> print(total_time / 10000)
0.0003786295175552368  # average time per loop
```

Which timer is `timeit` using? According to `timeit`'s source code, it uses the best timer available:

```
import sys

if sys.platform == 'win32':
    # On Windows, the best timer is time.clock
    default_timer = time.clock
else:
    # On most other platforms the best timer is time.time
    default_timer = time.time
```

Another important mechanism of `timeit` is that it disables the garbage collector during execution, as shown in the following code:

```
import gc

gcold = gc.isenabled()
gc.disable()
try:
    timing = self.inner(it, self.timer)
finally:
    if gcold:
        gc.enable()
```

If garbage collection should be enabled to measure the program's performance more

accurately, i.e., when the program allocates and de-allocates lots of objects, then you should enable it during the setup:

```
>>> timeit.timeit("[v for v in range(10000)]", setup="gc.enable()", number=10000)
3.6051759719848633
```

Except for very special cases, you should always use the module `timeit` to benchmark a program. In addition, it is valuable to remember that measuring the performance of a program is always context-dependent since no program is executing in a system with boundless computing resources and an average time measured from a number of loops is always better than one time measured in one execution.