time — Clock Time

Purpose: Functions for manipulating clock time.

The `time` module provides access to several different types of clocks, each useful for different purposes. The standard system calls like `time()` report the system "wall clock" time. The `monotonic()` clock can be used to measure elapsed time in a long-running process because it is guaranteed never to move backwards, even if the system time is changed. For performance testing, `perf_counter()` provides access to the clock with the highest available resolution to make short time measurements more accurate. The CPU time is available through `clock()`, and `process_time()` returns the combined processor time and system time.

Note

The implementations expose C library functions for manipulating dates and times. Since they are tied to the underlying C implementation, some details (such as the start of the epoch and maximum date value supported) are platform-specific. Refer to the library documentation for complete details.

Comparing Clocks

Implementation details for the clocks varies by platform. Use `get_clock_info()` to access basic information about the current implementation, including the clock's resolution.

```
# time_get_clock_info.py

import textwrap
import time

available_clocks = [
    ('monotonic', time.monotonic),
    ('perf_counter', time.perf_counter),
    ('process_time', time.process_time),
    ('time', time.time),
]

for clock_name, func in available_clocks:
    print(textwrap.dedent('''\
    {name}:
        adjustable    : {info.adjustable}
        implementation: {info.implementation}
        monotonic     : {info.monotonic}
        resolution    : {info.resolution}
        current       : {current}
    ''').format(
        name=clock_name,
        info=time.get_clock_info(clock_name),
        current=func())
    )
```

This output for macOS shows that the monotonic and perf_counter clocks are implemented using the same underlying system call.

```
$ python3 time_get_clock_info.py

monotonic:
    adjustable    : False
    implementation: mach_absolute_time()
    monotonic     : True
    resolution    : 1e-09
    current       : 0.047857746

perf_counter:
    adjustable    : False
```

```
    implementation: mach_absolute_time()
    monotonic     : True
    resolution    : 1e-09
    current       : 0.047930006

process_time:
    adjustable    : False
    implementation: getrusage(RUSAGE_SELF)
    monotonic     : True
    resolution    : 1e-06
    current       : 0.074122

time:
    adjustable    : True
    implementation: gettimeofday()
    monotonic     : False
    resolution    : 1e-06
    current       : 1544377423.803307
```

Wall Clock Time

One of the core functions of the `time` module is `time()`, which returns the number of seconds since the start of the "epoch" as a floating point value.

```
# time_time.py

import time

print('The time is:', time.time())
```

The epoch is the start of measurement for time, which for Unix systems is 0:00 on January 1, 1970. Although the value is always a float, actual precision is platform-dependent.

```
$ python3 time_time.py

The time is: 1544377423.849656
```

The float representation is useful when storing or comparing dates, but not as useful for producing human readable representations. For logging or printing time `ctime()` can be more useful.

```
# time_ctime.py

import time

print('The time is      :', time.ctime())
later = time.time() + 15
print('15 secs from now :', time.ctime(later))
```

The second `print()` call in this example shows how to use `ctime()` to format a time value other than the current time.

```
$ python3 time_ctime.py

The time is      : Sun Dec  9 12:43:43 2018
15 secs from now : Sun Dec  9 12:43:58 2018
```

Monotonic Clocks

Because `time()` looks at the system clock, and the system clock can be changed by the user or system services for synchronizing clocks across multiple computers, calling `time()` repeatedly may produce values that go forwards and backwards. This can result in unexpected behavior when trying to measure durations or otherwise use those times for computation. Avoid those situations by using `monotonic()`, which always returns values that go forward.

```
# time_monotonic.py

import time

start = time.monotonic()
```

```
start = time.monotonic()
time.sleep(0.1)
end = time.monotonic()
print('start : {:>9.2f}'.format(start))
print('end   : {:>9.2f}'.format(end))
print('span  : {:>9.2f}'.format(end - start))
```

The start point for the monotonic clock is not defined, so return values are only useful for doing calculations with other clock values. In this example the duration of the sleep is measured using `monotonic()`.

```
$ python3 time_monotonic.py

start :      0.02
end   :      0.13
span  :      0.10
```

Processor Clock Time

While `time()` returns a wall clock time, `process_time()` returns processor clock time. The values returned from `process_time()` reflect the actual time used by the program as it runs.

```
# time_process_time.py

import hashlib
import time

# Data to use to calculate md5 checksums
data = open(__file__, 'rb').read()

for i in range(5):
    h = hashlib.sha1()
    print(time.ctime(), ': {:0.3f} {:0.3f}'.format(
        time.time(), time.process_time()))
    for i in range(300000):
        h.update(data)
    cksum = h.digest()
```

In this example, the formatted `ctime()` is printed along with the floating point values from `time()`, and `clock()` for each iteration through the loop.

Note

If you want to run the example on your system, you may have to add more cycles to the inner loop or work with a larger amount of data to actually see a difference in the times.

```
$ python3 time_process_time.py

Sun Dec  9 12:43:44 2018 : 1544377424.095 0.056
Sun Dec  9 12:43:44 2018 : 1544377424.423 0.675
Sun Dec  9 12:43:44 2018 : 1544377424.790 1.376
Sun Dec  9 12:43:45 2018 : 1544377425.108 1.997
Sun Dec  9 12:43:45 2018 : 1544377425.422 2.594
```

Typically, the processor clock does not tick if a program is not doing anything.

```
# time_clock_sleep.py

import time

template = '{} - {:0.2f} - {:0.2f}'

print(template.format(
    time.ctime(), time.time(), time.process_time())
)

for i in range(3, 0, -1):
    print('Sleeping', i)
```

```
    time.sleep(i)
    print(template.format(
        time.ctime(), time.time(), time.process_time())
    )
```

In this example, the loop does very little work by going to sleep after each iteration. The `time()` value increases even while the application is asleep, but the `process_time()` value does not.

```
$ python3 -u time_clock_sleep.py

Sun Dec  9 12:43:45 2018 - 1544377425.83 - 0.06
Sleeping 3
Sun Dec  9 12:43:48 2018 - 1544377428.83 - 0.06
Sleeping 2
Sun Dec  9 12:43:50 2018 - 1544377430.84 - 0.06
Sleeping 1
Sun Dec  9 12:43:51 2018 - 1544377431.84 - 0.06
```

Calling `sleep()` yields control from the current thread and asks it to wait for the system to wake it back up. If a program has only one thread, this effectively blocks the app and it does no work.

Performance Counter

It is important to have a high-resolution monotonic clock for measuring performance. Determining the best clock data source requires platform-specific knowledge, which Python provides in `perf_counter()`.

```
# time_perf_counter.py

import hashlib
import time

# Data to use to calculate md5 checksums
data = open(__file__, 'rb').read()

loop_start = time.perf_counter()

for i in range(5):
    iter_start = time.perf_counter()
    h = hashlib.sha1()
    for i in range(300000):
        h.update(data)
    cksum = h.digest()
    now = time.perf_counter()
    loop_elapsed = now - loop_start
    iter_elapsed = now - iter_start
    print(time.ctime(), ': {:0.3f} {:0.3f}'.format(
        iter_elapsed, loop_elapsed))
```

As with `monotonic()`, the epoch for `perf_counter()` is undefined, and the values are meant to be used for comparing and computing values, not as absolute times.

```
$ python3 time_perf_counter.py

Sun Dec  9 12:43:52 2018 : 0.366 0.366
Sun Dec  9 12:43:52 2018 : 0.338 0.704
Sun Dec  9 12:43:52 2018 : 0.350 1.054
Sun Dec  9 12:43:53 2018 : 0.315 1.369
Sun Dec  9 12:43:53 2018 : 0.306 1.675
```

Time Components

Storing times as elapsed seconds is useful in some situations, but there are times when a program needs to have access to the individual fields of a date (year, month, etc.). The `time` module defines `struct_time` for holding date and time values with components broken out so they are easy to access. There are several functions that work with `struct_time` values instead of floats.

```
# time_struct.py
```

```
import time


def show_struct(s):
    print('  tm_year :', s.tm_year)
    print('  tm_mon  :', s.tm_mon)
    print('  tm_mday :', s.tm_mday)
    print('  tm_hour :', s.tm_hour)
    print('  tm_min  :', s.tm_min)
    print('  tm_sec  :', s.tm_sec)
    print('  tm_wday :', s.tm_wday)
    print('  tm_yday :', s.tm_yday)
    print('  tm_isdst:', s.tm_isdst)


print('gmtime:')
show_struct(time.gmtime())
print('\nlocaltime:')
show_struct(time.localtime())
print('\nmktime:', time.mktime(time.localtime()))
```

The `gmtime()` function returns the current time in UTC. `localtime()` returns the current time with the current time zone applied. `mktime()` takes a `struct_time` and converts it to the floating point representation.

```
$ python3 time_struct.py

gmtime:
  tm_year : 2018
  tm_mon  : 12
  tm_mday : 9
  tm_hour : 17
  tm_min  : 43
  tm_sec  : 53
  tm_wday : 6
  tm_yday : 343
  tm_isdst: 0

localtime:
  tm_year : 2018
  tm_mon  : 12
  tm_mday : 9
  tm_hour : 12
  tm_min  : 43
  tm_sec  : 53
  tm_wday : 6
  tm_yday : 343
  tm_isdst: 0

mktime: 1544377433.0
```

Working with Time Zones

The functions for determining the current time depend on having the time zone set, either by the program or by using a default time zone set for the system. Changing the time zone does not change the actual time, just the way it is represented.

To change the time zone, set the environment variable TZ, then call `tzset()`. The time zone can be specified with a lot of detail, right down to the start and stop times for daylight savings time. It is usually easier to use the time zone name and let the underlying libraries derive the other information, though.

This example program changes the time zone to a few different values and shows how the changes affect other settings in the time module.

```
# time_timezone.py
```

```
import time
import os


def show_zone_info():
```

```
    print('  TZ    :', os.environ.get('TZ', '(not set)'))
    print('  tzname:', time.tzname)
    print('  Zone  : {} ({})'.format(
        time.timezone, (time.timezone / 3600)))
    print('  DST   :', time.daylight)
    print('  Time  :', time.ctime())
    print()


print('Default :')
show_zone_info()

ZONES = [
    'GMT',
    'Europe/Amsterdam',
]

for zone in ZONES:
    os.environ['TZ'] = zone
    time.tzset()
    print(zone, ':')
    show_zone_info()
```

The default time zone on the system used to prepare the examples is US/Eastern. The other zones in the example change the tzname, daylight flag, and timezone offset value.

```
$ python3 time_timezone.py

Default :
  TZ    : (not set)
  tzname: ('EST', 'EDT')
  Zone  : 18000 (5.0)
  DST   : 1
  Time  : Sun Dec  9 12:43:53 2018

GMT :
  TZ    : GMT
  tzname: ('GMT', 'GMT')
  Zone  : 0 (0.0)
  DST   : 0
  Time  : Sun Dec  9 17:43:53 2018

Europe/Amsterdam :
  TZ    : Europe/Amsterdam
  tzname: ('CET', 'CEST')
  Zone  : -3600 (-1.0)
  DST   : 1
  Time  : Sun Dec  9 18:43:53 2018
```

Parsing and Formatting Times

The two functions `strptime()` and `strftime()` convert between `struct_time` and string representations of time values. There is a long list of formatting instructions available to support input and output in different styles. The complete list is documented in the library documentation for the `time` module.

This example converts the current time from a string to a `struct_time` instance and back to a string.

```
# time_strptime.py

import time


def show_struct(s):
    print('  tm_year :', s.tm_year)
    print('  tm_mon  :', s.tm_mon)
    print('  tm_mday :', s.tm_mday)
    print('  tm_hour :', s.tm_hour)
    print('  tm_min  :', s.tm_min)
    print('  tm_sec  :', s.tm_sec)
    print('  tm_wday :', s.tm_wday)
```

```
    print('  tm_yday :', s.tm_yday)
    print('  tm_isdst:', s.tm_isdst)


now = time.ctime(1483391847.433716)
print('Now:', now)

parsed = time.strptime(now)
print('\nParsed:')
show_struct(parsed)

print('\nFormatted:',
      time.strftime("%a %b %d %H:%M:%S %Y", parsed))
```

The output string is not exactly like the input, since the day of the month is prefixed with a zero.

```
$ python3 time_strptime.py

Now: Mon Jan  2 16:17:27 2017

Parsed:
  tm_year : 2017
  tm_mon  : 1
  tm_mday : 2
  tm_hour : 16
  tm_min  : 17
  tm_sec  : 27
  tm_wday : 0
  tm_yday : 2
  tm_isdst: -1

Formatted: Mon Jan 02 16:17:27 2017
```

See also

- Standard library documentation for time
- Python 2 to 3 porting notes for time
- `datetime` – The `datetime` module includes other classes for doing calculations with dates and times.
- `calendar` – Work with higher-level date functions to produce calendars or calculate recurring events.

Dates and Times

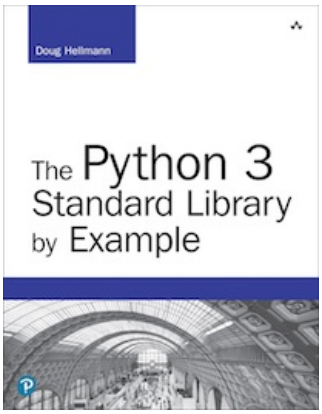
datetime — Date and Time Value Manipulation

Quick Links

This page was last updated 2018-12-09.

Navigation

Dates and Times
datetime — Date and Time Value Manipulation



Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example