A Beginner's Guide to the Python time Module

realpython.com/python-time-module

About Alex Ronquillo Alex Ronquillo is a Software Engineer at thelab. He's an avid Pythonista who is also passionate about writing and game development. » More about Alex Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are: Aldren Brad Joanna

> Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Mastering Python's Built-in time Module**

The Python `time` module provides many ways of representing time in code, such as objects, numbers, and strings. It also provides functionality other than representing time, like waiting during code execution and measuring the efficiency of your code.

This article will walk you through the most commonly used functions and objects in `time`.

By the end of this article, you'll be able to:

- **Understand** core concepts at the heart of working with dates and times, such as epochs, time zones, and daylight savings time
- **Represent** time in code using floats, tuples, and `struct_time`
- **Convert** between different time representations
- **Suspend** thread execution
- **Measure** code performance using `perf_counter()`

You'll start by learning how you can use a floating point number to represent time.

Free Bonus: Click here to get our free Python Cheat Sheet that shows you the basics of Python 3, like working with data types, dictionaries, lists, and Python functions.

Dealing With Python Time Using Seconds

One of the ways you can manage the concept of Python time in your application is by using a floating point number that represents the number of seconds that have passed since the beginning of an era—that is, since a certain starting point.

Let's dive deeper into what that means, why it's useful, and how you can use it to implement logic, based on Python time, in your application.

The Epoch

You learned in the previous section that you can manage Python time with a floating point number representing elapsed time since the beginning of an era.

Merriam-Webster defines an era as:

- A fixed point in time from which a series of years is reckoned
- A system of chronological notation computed from a given date as basis

The important concept to grasp here is that, when dealing with Python time, you're considering a period of time identified by a starting point. In computing, you call this starting point the **epoch**.

The epoch, then, is the starting point against which you can measure the passage of time.

For example, if you define the epoch to be midnight on January 1, 1970 UTC—the epoch as defined on Windows and most UNIX systems—then you can represent midnight on January 2, 1970 UTC as `86400` seconds since the epoch.

This is because there are 60 seconds in a minute, 60 minutes in an hour, and 24 hours in a day. January 2, 1970 UTC is only one day after the epoch, so you can apply basic math to arrive at that result:

```
>>>
```

```
>>> 60 * 60 * 24
86400
```

It is also important to note that you can still represent time before the epoch. The number of seconds would just be negative.

For example, you would represent midnight on December 31, 1969 UTC (using an epoch of January 1, 1970) as `-86400` seconds.

While January 1, 1970 UTC is a common epoch, it is not the only epoch used in computing. In fact, different operating systems, filesystems, and APIs sometimes use different epochs.

As you saw before, UNIX systems define the epoch as January 1, 1970. The Win32 API, on the other hand, defines the epoch as January 1, 1601.

You can use `time.gmtime()` to determine your system's epoch:

```
>>>
```

```
>>> import time
>>> time.gmtime(0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=3, tm_yday=1, tm_isdst=0)
```

You'll learn about `gmtime()` and `struct_time` throughout the course of this article. For now, just know that you can use `time` to discover the epoch using this function.

Now that you understand more about how to measure time in seconds using an epoch, let's take a look at Python's `time` module to see what functions it offers that help you do so.

Python Time in Seconds as a Floating Point Number

First, `time.time()` returns the number of seconds that have passed since the epoch. The return value is a floating point number to account for fractional seconds:

```
>>>
```

```
>>> from time import time
>>> time()
1551143536.9323719
```

The number you get on your machine may be very different because the reference point considered to be the epoch may be very different.

Further Reading: Python 3.7 introduced time_ns(), which returns an integer value representing the same elapsed time since the epoch, but in nanoseconds rather than seconds.

Measuring time in seconds is useful for a number of reasons:

- You can use a float to calculate the difference between two points in time.
- A float is easily serializable, meaning that it can be stored for data transfer and come out intact on the other side.

Sometimes, however, you may want to see the current time represented as a string. To do so, you can pass the number of seconds you get from `time()` into `time.ctime()`.

Python Time in Seconds as a String Representing Local Time

As you saw before, you may want to convert the Python time, represented as the number of elapsed seconds since the epoch, to a string. You can do so using `ctime()`:

```
>>>
```

```
>>> from time import time, ctime
>>> t = time()
>>> ctime(t)
'Mon Feb 25 19:11:59 2019'
```

Here, you've recorded the current time in seconds into the variable `t`, then passed `t` as an argument to `ctime()`, which returns a string representation of that same time.

Technical Detail: The argument, representing seconds since the epoch, is optional according to the `ctime()` definition. If you don't pass an argument, then `ctime()` uses the return value of `time()` by default. So, you could simplify the example above:

```
>>>
```

```
>>> from time import ctime
>>> ctime()
'Mon Feb 25 19:11:59 2019'
```

The string representation of time, also known as a **timestamp**, returned by `ctime()` is formatted with the following structure:

1. **Day of the week:** `Mon` (`Monday`)
2. **Month of the year:** `Feb` (`February`)
3. **Day of the month:** `25`
4. **Hours, minutes, and seconds using the 24-hour clock notation:** `19:11:59`
5. **Year:** `2019`

The previous example displays the timestamp of a particular moment captured from a computer in the South Central region of the United States. But, let's say you live in Sydney, Australia, and you executed the same command at the same instant.

Instead of the above output, you'd see the following:

```
>>>
```

```
>>> from time import time, ctime
>>> t = time()
>>> ctime(t)
'Tue Feb 26 12:11:59 2019'
```

Notice that the `day of week` , `day of month` , and `hour` portions of the timestamp are different than the first example.

These outputs are different because the timestamp returned by `ctime()` depends on your geographical location.

Note: While the concept of time zones is relative to your physical location, you can modify this in your computer's settings without actually relocating.

The representation of time dependent on your physical location is called **local time** and makes use of a concept called **time zones**.

Note: Since local time is related to your locale, timestamps often account for locale-specific details such as the order of the elements in the string and translations of the day and month abbreviations. `ctime()` ignores these details.

Let's dig a little deeper into the notion of time zones so that you can better understand Python time representations.

Understanding Time Zones

A time zone is a region of the world that conforms to a standardized time. Time zones are defined by their offset from Coordinated Universal Time (UTC) and, potentially, the inclusion of daylight savings time (which we'll cover in more detail later in this article).

Fun Fact: If you're a native English speaker, you might be wondering why the abbreviation for "Coordinated Universal Time" is UTC rather than the more obvious CUT. However, if you're a native French speaker, you would call it "Temps Universel Coordonné," which suggests a different abbreviation: TUC.

Ultimately, the International Telecommunication Union and the International Astronomical Union compromised on UTC as the official abbreviation so that, regardless of language, the abbreviation would be the same.

UTC and Time Zones

UTC is the time standard against which all the world's timekeeping is synchronized (or coordinated). It is not, itself, a time zone but rather a transcendent standard that defines what time zones are.

UTC time is precisely measured using astronomical time, referring to the Earth's rotation, and atomic clocks.

Time zones are then defined by their offset from UTC. For example, in North and South America, the Central Time Zone (CT) is behind UTC by five or six hours and, therefore, uses the notation UTC-5:00 or UTC-6:00.

Sydney, Australia, on the other hand, belongs to the Australian Eastern Time Zone (AET), which is ten or eleven hours ahead of UTC (UTC+10:00 or UTC+11:00).

This difference (UTC-6:00 to UTC+10:00) is the reason for the variance you observed in the two outputs from `ctime()` in the previous examples:

- **Central Time (CT):** `'Mon Feb 25 19:11:59 2019'`
- **Australian Eastern Time (AET):** `'Tue Feb 26 12:11:59 2019'`

These times are exactly sixteen hours apart, which is consistent with the time zone offsets mentioned above.

You may be wondering why CT can be either five or six hours behind UTC or why AET can be ten or eleven hours ahead. The reason for this is that some areas around the world, including parts of these time zones, observe daylight savings time.

Daylight Savings Time

Summer months generally experience more daylight hours than winter months. Because of this, some areas observe daylight savings time (DST) during the spring and summer to make better use of those daylight hours.

For places that observe DST, their clocks will jump ahead one hour at the beginning of spring (effectively losing an hour). Then, in the fall, the clocks will be reset to standard time.

The letters S and D represent standard time and daylight savings time in time zone notation:

- Central Standard Time (CST)
- Australian Eastern Daylight Time (AEDT)

When you represent times as timestamps in local time, it is always important to consider whether DST is applicable or not.

`ctime()` accounts for daylight savings time. So, the output difference listed previously would be more accurate as the following:

- **Central Standard Time (CST):** `'Mon Feb 25 19:11:59 2019'`
- **Australian Eastern Daylight Time (AEDT):** `'Tue Feb 26 12:11:59 2019'`

Dealing With Python Time Using Data Structures

Now that you have a firm grasp on many fundamental concepts of time including epochs, time zones, and UTC, let's take a look at more ways to represent time using the Python `time` module.

Python Time as a Tuple

Instead of using a number to represent Python time, you can use another primitive data structure: a tuple.

The tuple allows you to manage time a little more easily by abstracting some of the data and making it more readable.

When you represent time as a tuple, each element in your tuple corresponds to a specific element of time:

1. Year
2. Month as an integer, ranging between 1 (January) and 12 (December)
3. Day of the month
4. Hour as an integer, ranging between 0 (12 A.M.) and 23 (11 P.M.)
5. Minute
6. Second
7. Day of the week as an integer, ranging between 0 (Monday) and 6 (Sunday)
8. Day of the year

9. Daylight savings time as an integer with the following values:
    - `1` is daylight savings time.
    - `0` is standard time.
    - `-1` is unknown.

Using the methods you've already learned, you can represent the same Python time in two different ways:

```
>>>

>>> from time import time, ctime
>>> t = time()
>>> t
1551186415.360564
>>> ctime(t)
'Tue Feb 26 07:06:55 2019'

>>> time_tuple = (2019, 2, 26, 7, 6, 55, 1, 57, 0)
```

In this case, both `t` and `time_tuple` represent the same time, but the tuple provides a more readable interface for working with time components.

Technical Detail: Actually, if you look at the Python time represented by `time_tuple` in seconds (which you'll see how to do later in this article), you'll see that it resolves to `1551186415.0` rather than `1551186415.360564`.

This is because the tuple doesn't have a way to represent fractional seconds.

While the tuple provides a more manageable interface for working with Python time, there is an even better object: `struct_time`.

Python Time as an Object

The problem with the tuple construct is that it still looks like a bunch of numbers, even though it's better organized than a single, seconds-based number.

`struct_time` provides a solution to this by utilizing NamedTuple, from Python's `collections` module, to associate the tuple's sequence of numbers with useful identifiers:

```
>>>

>>> from time import struct_time
>>> time_tuple = (2019, 2, 26, 7, 6, 55, 1, 57, 0)
>>> time_obj = struct_time(time_tuple)
>>> time_obj
time.struct_time(tm_year=2019, tm_mon=2, tm_mday=26, tm_hour=7, tm_min=6,
tm_sec=55, tm_wday=1, tm_yday=57, tm_isdst=0)
```

Technical Detail: If you're coming from another language, the terms `struct` and `object` might be in opposition to one another.

In Python, there is no data type called `struct`. Instead, everything is an object.

However, the name `struct_time` is derived from the C-based time library where the data type is actually a `struct`.

In fact, Python's `time` module, which is implemented in C, uses this `struct` directly by including the header file `times.h`.

Now, you can access specific elements of `time_obj` using the attribute's name rather than an index:

```
>>>

>>> day_of_year = time_obj.tm_yday
>>> day_of_year
57
>>> day_of_month = time_obj.tm_mday
>>> day_of_month
26
```

Beyond the readability and usability of `struct_time`, it is also important to know because it is the return type of many of the functions in the Python `time` module.

Converting Python Time in Seconds to an Object

Now that you've seen the three primary ways of working with Python time, you'll learn how to convert between the different time data types.

Converting between time data types is dependent on whether the time is in UTC or local time.

Coordinated Universal Time (UTC)

The epoch uses UTC for its definition rather than a time zone. Therefore, the seconds elapsed since the epoch is not variable depending on your geographical location.

However, the same cannot be said of `struct_time`. The object representation of Python time may or may not take your time zone into account.

There are two ways to convert a float representing seconds to a `struct_time`:

1. UTC
2. Local time

To convert a Python time float to a UTC-based `struct_time`, the Python `time` module provides a function called `gmtime()`.

You've seen `gmtime()` used once before in this article:

>>>

```
>>> import time
>>> time.gmtime(0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=3, tm_yday=1, tm_isdst=0)
```

You used this call to discover your system's epoch. Now, you have a better foundation for understanding what's actually happening here.

`gmtime()` converts the number of elapsed seconds since the epoch to a `struct_time` in UTC. In this case, you've passed `0` as the number of seconds, meaning you're trying to find the epoch, itself, in UTC.

Note: Notice the attribute `tm_isdst` is set to `0`. This attribute represents whether the time zone is using daylight savings time. UTC never subscribes to DST, so that flag will always be `0` when using `gmtime()`.

As you saw before, `struct_time` cannot represent fractional seconds, so `gmtime()` ignores the fractional seconds in the argument:

>>>

```
>>> import time
>>> time.gmtime(1.99)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=1,
tm_wday=3, tm_yday=1, tm_isdst=0)
```

Notice that even though the number of seconds you passed was very close to `2`, the `.99` fractional seconds were simply ignored, as shown by `tm_sec=1`.

The `secs` parameter for `gmtime()` is optional, meaning you can call `gmtime()` with no arguments. Doing so will provide the current time in UTC:

>>>

```
>>> import time
>>> time.gmtime()
time.struct_time(tm_year=2019, tm_mon=2, tm_mday=28, tm_hour=12, tm_min=57,
tm_sec=24, tm_wday=3, tm_yday=59, tm_isdst=0)
```

Interestingly, there is no inverse for this function within `time`. Instead, you'll have to look in Python's `calendar` module for a function named `timegm()`:

>>>

```
>>> import calendar
>>> import time
>>> time.gmtime()
time.struct_time(tm_year=2019, tm_mon=2, tm_mday=28, tm_hour=13, tm_min=23,
tm_sec=12, tm_wday=3, tm_yday=59, tm_isdst=0)
>>> calendar.timegm(time.gmtime())
1551360204
```

`timegm()` takes a tuple (or `struct_time`, since it is a subclass of tuple) and returns the corresponding number of seconds since the epoch.

Historical Context: If you're interested in why `timegm()` is not in `time`, you can view the discussion in Python Issue 6280.

In short, it was originally added to `calendar` because `time` closely follows C's time library (defined in `time.h`), which contains no matching function. The above-mentioned issue proposed the idea of moving or copying `timegm()` into `time`.

However, with advances to the `datetime` library, inconsistencies in the patched implementation of `time.timegm()`, and a question of how to then handle `calendar.timegm()`, the maintainers declined the patch, encouraging the use of `datetime` instead.

Working with UTC is valuable in programming because it's a standard. You don't have to worry about DST, time zone, or locale information.

That said, there are plenty of cases when you'd want to use local time. Next, you'll see how to convert from seconds to local time so that you can do just that.

Local Time

In your application, you may need to work with local time rather than UTC. Python's `time` module provides a function for getting local time from the number of seconds elapsed since the epoch called `localtime()`.

The signature of `localtime()` is similar to `gmtime()` in that it takes an optional `secs` argument, which it uses to build a `struct_time` using your local time zone:

```
>>>

>>> import time
>>> time.time()
1551448206.86196
>>> time.localtime(1551448206.86196)
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=1, tm_hour=7, tm_min=50, tm_sec=6,
tm_wday=4, tm_yday=60, tm_isdst=0)
```

Notice that `tm_isdst=0`. Since DST matters with local time, `tm_isdst` will change between `0` and `1` depending on whether or not DST is applicable for the given time. Since `tm_isdst=0`, DST is not applicable for March 1, 2019.

In the United States in 2019, daylight savings time begins on March 10. So, to test if the DST flag will change correctly, you need to add 9 days' worth of seconds to the `secs` argument.

To compute this, you take the number of seconds in a day (86,400) and multiply that by 9 days:

```
>>>

>>> new_secs = 1551448206.86196 + (86400 * 9)
>>> time.localtime(new_secs)
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=10, tm_hour=8, tm_min=50,
tm_sec=6, tm_wday=6, tm_yday=69, tm_isdst=1)
```

Now, you'll see that the `struct_time` shows the date to be March 10, 2019 with `tm_isdst=1`. Also, notice that `tm_hour` has also jumped ahead, to `8` instead of `7` in the previous example, because of daylight savings time.

Since Python 3.3, `struct_time` has also included two attributes that are useful in determining the time zone of the `struct_time`:

1. `tm_zone`
2. `tm_gmtoff`

At first, these attributes were platform dependent, but they have been available on all platforms since Python 3.6.

First, `tm_zone` stores the local time zone:

```
>>>

>>> import time
>>> current_local = time.localtime()
>>> current_local.tm_zone
'CST'
```

Here, you can see that `localtime()` returns a `struct_time` with the time zone set to `CST` (Central Standard Time).

As you saw before, you can also tell the time zone based on two pieces of information, the UTC offset and DST (if applicable):

```
>>>

>>> import time
>>> current_local = time.localtime()
>>> current_local.tm_gmtoff
-21600
>>> current_local.tm_isdst
0
```

In this case, you can see that `current_local` is `21600` seconds behind GMT, which stands for Greenwich Mean Time. GMT is the time zone with no UTC offset: UTC±00:00.

`21600` seconds divided by seconds per hour (3,600) means that `current_local` time is `GMT-06:00` (or `UTC-06:00`).

You can use the GMT offset plus the DST status to deduce that `current_local` is `UTC-06:00` at standard time, which corresponds to the Central standard time zone.

Like `gmtime()`, you can ignore the `secs` argument when calling `localtime()`, and it will return the current local time in a `struct_time`:

```
>>>

>>> import time
>>> time.localtime()
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=1, tm_hour=8, tm_min=34,
tm_sec=28, tm_wday=4, tm_yday=60, tm_isdst=0)
```

Unlike `gmtime()`, the inverse function of `localtime()` does exist in the Python `time` module. Let's take a look at how that works.

Converting a Local Time Object to Seconds

You've already seen how to convert a UTC time object to seconds using `calendar.timegm()`. To convert local time to seconds, you'll use `mktime()`.

`mktime()` requires you to pass a parameter called `t` that takes the form of either a normal 9-tuple or a `struct_time` object representing local time:

```
>>>

>>> import time

>>> time_tuple = (2019, 3, 10, 8, 50, 6, 6, 69, 1)
>>> time.mktime(time_tuple)
1552225806.0

>>> time_struct = time.struct_time(time_tuple)
>>> time.mktime(time_struct)
1552225806.0
```

It's important to keep in mind that `t` must be a tuple representing local time, not UTC:

```
>>>
```

```
>>> from time import gmtime, mktime

>>> # 1
>>> current_utc = time.gmtime()
>>> current_utc
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=1, tm_hour=14, tm_min=51,
tm_sec=19, tm_wday=4, tm_yday=60, tm_isdst=0)

>>> # 2
>>> current_utc_secs = mktime(current_utc)
>>> current_utc_secs
1551473479.0

>>> # 3
>>> time.gmtime(current_utc_secs)
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=1, tm_hour=20, tm_min=51,
tm_sec=19, tm_wday=4, tm_yday=60, tm_isdst=0)
```

Note: For this example, assume that the local time is `March 1, 2019 08:51:19 CST`.

This example shows why it's important to use `mktime()` with local time, rather than UTC:

1. **`gmtime()`** with no argument returns a `struct_time` using UTC. `current_utc` shows `March 1, 2019 14:51:19 UTC`. This is accurate because `CST is UTC-06:00`, so UTC should be 6 hours ahead of local time.

2. **`mktime()`** tries to return the number of seconds, expecting local time, but you passed `current_utc` instead. So, instead of understanding that `current_utc` is UTC time, it assumes you meant `March 1, 2019 14:51:19 CST`.

3. **`gmtime()`** is then used to convert those seconds back into UTC, which results in an inconsistency. The time is now `March 1, 2019 20:51:19 UTC`. The reason for this discrepancy is the fact that `mktime()` expected local time. So, the conversion back to UTC adds *another* 6 hours to local time.

Working with time zones is notoriously difficult, so it's important to set yourself up for success by understanding the differences between UTC and local time and the Python time functions that deal with each.

Converting a Python Time Object to a String

While working with tuples is fun and all, sometimes it's best to work with strings.

String representations of time, also known as timestamps, help make times more readable and can be especially useful for building intuitive user interfaces.

There are two Python `time` functions that you use for converting a `time.struct_time` object to a string:

1. `asctime()`
2. `strftime()`

You'll begin by learning about `asctime()`.

asctime()

You use `asctime()` for converting a time tuple or `struct_time` to a timestamp:

```
>>>
```

```
>>> import time
>>> time.asctime(time.gmtime())
'Fri Mar  1 18:42:08 2019'
>>> time.asctime(time.localtime())
'Fri Mar  1 12:42:15 2019'
```

Both `gmtime()` and `localtime()` return `struct_time` instances, for UTC and local time respectively.

You can use `asctime()` to convert either `struct_time` to a timestamp. `asctime()` works similarly to `ctime()`, which you learned about earlier in this article, except instead of passing a floating point number, you pass a tuple. Even the timestamp format is the same between the two functions.

As with `ctime()`, the parameter for `asctime()` is optional. If you do not pass a time object to `asctime()`, then it will use the current local time:

```
>>>
```

```
>>> import time
>>> time.asctime()
'Fri Mar  1 12:56:07 2019'
```

As with `ctime()`, it also ignores locale information.

One of the biggest drawbacks of `asctime()` is its format inflexibility. `strftime()` solves this problem by allowing you to format your timestamps.

strftime()

You may find yourself in a position where the string format from `ctime()` and `asctime()` isn't satisfactory for your application. Instead, you may want to format your strings in a way that's more meaningful to your users.

One example of this is if you would like to display your time in a string that takes locale information into account.

To format strings, given a `struct_time` or Python time tuple, you use `strftime()`, which stands for "string **format** time."

`strftime()` takes two arguments:

1. **`format`** specifies the order and form of the time elements in your string.
2. **`t`** is an optional time tuple.

To format a string, you use **directives**. Directives are character sequences that begin with a `%` that specify a particular time element, such as:

- **`%d`** **:** Day of the month
- **`%m`** **:** Month of the year
- **`%Y`** **:** Year

For example, you can output the date in your local time using the ISO 8601 standard like this:

>>>

```
>>> import time
>>> time.strftime('%Y-%m-%d', time.localtime())
'2019-03-01'
```

Further Reading: While representing dates using Python time is completely valid and acceptable, you should also consider using Python's `datetime` module, which provides shortcuts and a more robust framework for working with dates and times together.

For example, you can simplify outputting a date in the ISO 8601 format using `datetime`:

>>>

```
>>> from datetime import date
>>> date(year=2019, month=3, day=1).isoformat()
'2019-03-01'
```

To learn more about using the Python `datetime` module, check out Using Python datetime to Work With Dates and Times

As you saw before, a great benefit of using `strftime()` over `asctime()` is its ability to render timestamps that make use of locale-specific information.

For example, if you want to represent the date and time in a locale-sensitive way, you can't use `asctime()`:

>>>

```
>>> from time import asctime
>>> asctime()
'Sat Mar  2 15:21:14 2019'

>>> import locale
>>> locale.setlocale(locale.LC_TIME, 'zh_HK')  # Chinese - Hong Kong
'zh_HK'
>>> asctime()
'Sat Mar  2 15:58:49 2019'
```

Notice that even after programmatically changing your locale, `asctime()` still returns the date and time in the same format as before.

Technical Detail: `LC_TIME` is the locale category for date and time formatting. The `locale` argument `'zh_HK'` may be different, depending on your system.

When you use `strftime()`, however, you'll see that it accounts for locale:

```
>>>

>>> from time import strftime, localtime
>>> strftime('%c', localtime())
'Sat Mar  2 15:23:20 2019'

>>> import locale
>>> locale.setlocale(locale.LC_TIME, 'zh_HK')  # Chinese - Hong Kong
'zh_HK'
>>> strftime('%c', localtime())
'六  3/ 2 15:58:12 2019' 2019'
```

Here, you have successfully utilized the locale information because you used `strftime()`.

Note: `%c` is the directive for locale-appropriate date and time.

If the time tuple is not passed to the parameter `t`, then `strftime()` will use the result of `localtime()` by default. So, you could simplify the examples above by removing the optional second argument:

```
>>>

>>> from time import strftime
>>> strftime('The current local datetime is: %c')
'The current local datetime is: Fri Mar  1 23:18:32 2019'
```

Here, you've used the default time instead of passing your own as an argument. Also, notice that the `format` argument can consist of text other than formatting directives.

Further Reading: Check out this thorough list of directives available to `strftime()`.

The Python `time` module also includes the inverse operation of converting a timestamp back into a `struct_time` object.

Converting a Python Time String to an Object

When you're working with date and time related strings, it can be very valuable to convert the timestamp to a time object.

To convert a time string to a `struct_time`, you use `strptime()`, which stands for "string **parse** time":

```
>>>

>>> from time import strptime
>>> strptime('2019-03-01', '%Y-%m-%d')
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=4, tm_yday=60, tm_isdst=-1)
```

The first argument to `strptime()` must be the timestamp you wish to convert. The second argument is the `format` that the timestamp is in.

The `format` parameter is optional and defaults to `'%a %b %d %H:%M:%S %Y'`. Therefore, if you have a timestamp in that format, you don't need to pass it as an argument:

```
>>>

>>> strptime('Fri Mar 01 23:38:40 2019')
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=1, tm_hour=23, tm_min=38,
tm_sec=40, tm_wday=4, tm_yday=60, tm_isdst=-1)
```

Since a `struct_time` has 9 key date and time components, `strptime()` must provide reasonable defaults for values for those components it can't parse from `string`.

In the previous examples, `tm_isdst=-1`. This means that `strptime()` can't determine by the timestamp whether it represents daylight savings time or not.

Now you know how to work with Python times and dates using the `time` module in a variety of ways. However, there are other uses for `time` outside of simply creating time objects, getting Python time strings, and using seconds elapsed since the epoch.

Suspending Execution

One really useful Python time function is `sleep()`, which suspends the thread's execution for a specified amount of time.

For example, you can suspend your program's execution for 10 seconds like this:

```
>>>
```

```
>>> from time import sleep, strftime
>>> strftime('%c')
'Fri Mar  1 23:49:26 2019'
>>> sleep(10)
>>> strftime('%c')
'Fri Mar  1 23:49:36 2019'
```

Your program will print the first formatted `datetime` string, then pause for 10 seconds, and finally print the second formatted `datetime` string.

You can also pass fractional seconds to `sleep()`:

```
>>>

>>> from time import sleep
>>> sleep(0.5)
```

`sleep()` is useful for testing or making your program wait for any reason, but you must be careful not to halt your production code unless you have good reason to do so.

Before Python 3.5, a signal sent to your process could interrupt `sleep()`. However, in 3.5 and later, `sleep()` will always suspend execution for at least the amount of specified time, even if the process receives a signal.

`sleep()` is just one Python time function that can help you test your programs and make them more robust.

Measuring Performance

You can use `time` to measure the performance of your program.

The way you do this is to use `perf_counter()` which, as the name suggests, provides a performance counter with a high resolution to measure short distances of time.

To use `perf_counter()`, you place a counter before your code begins execution as well as after your code's execution completes:

```
>>>

>>> from time import perf_counter
>>> def longrunning_function():
...     for i in range(1, 11):
...         time.sleep(i / i ** 2)
...
>>> start = perf_counter()
>>> longrunning_function()
>>> end = perf_counter()
>>> execution_time = (end - start)
>>> execution_time
8.201258441999926
```

First, `start` captures the moment before you call the function. `end` captures the moment after the function returns. The function's total execution time took `(end - start)` seconds.

Technical Detail: Python 3.7 introduced `perf_counter_ns()`, which works the same as `perf_counter()`, but uses nanoseconds instead of seconds.

`perf_counter()` (or `perf_counter_ns()`) is the most precise way to measure the performance of your code using one execution. However, if you're trying to accurately gauge the performance of a code snippet, I recommend using the Python timeit module.

`timeit` specializes in running code many times to get a more accurate performance analysis and helps you to avoid oversimplifying your time measurement as well as other common pitfalls.

Conclusion

Congratulations! You now have a great foundation for working with dates and times in Python.

Now, you're able to:

- Use a floating point number, representing seconds elapsed since the epoch, to deal with time
- Manage time using tuples and `struct_time` objects
- Convert between seconds, tuples, and timestamp strings
- Suspend the execution of a Python thread
- Measure performance using `perf_counter()`

On top of all that, you've learned some fundamental concepts surrounding date and time, such as:

- Epochs
- UTC
- Time zones
- Daylight savings time

Now, it's time for you to apply your newfound knowledge of Python time in your real world applications!

Further Reading

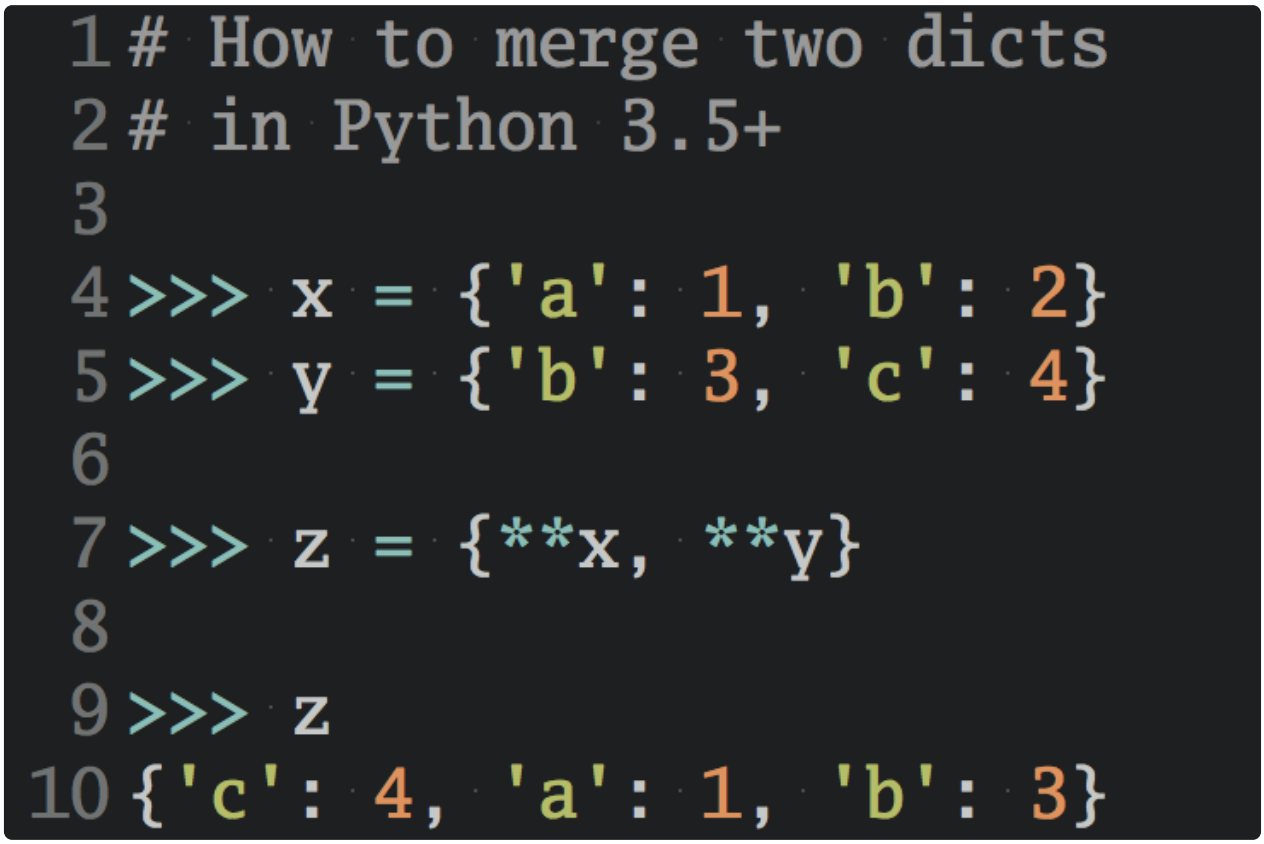
If you want to continue learning more about using dates and times in Python, take a look at the following modules:

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding:

Mastering Python's Built-in time Module

🐍 Python Tricks 💌

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.



About **Alex Ronquillo**

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

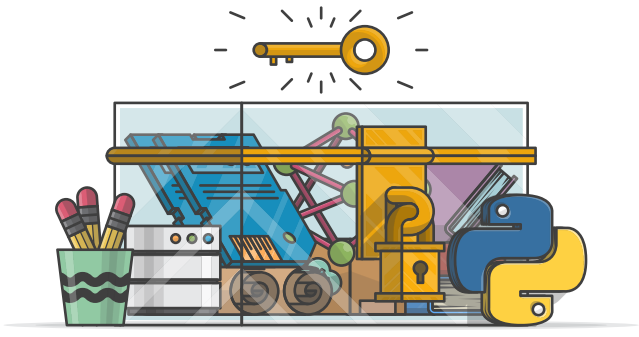
Aldren

Brad

Joanna

Master Real-World Python Skills With Unlimited Access to Real Python

Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:



Level Up Your Python Skills »

What Do You Think?

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here. What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.