Chapter 13. Calendars and Clocks

> *"One!" strikes the clock in the belfry tower,*
> *Which but sixty minutes ago*
> *Sounded twelve for the midnight hour.*
>
> —Frederick B. Needham, The Round of the Clock

> *I've been on a calendar but I have never been on time.*
>
> —Marilyn Monroe

Programmers devote a surprising amount of effort to dates and times. Let's talk about some of the problems they encounter and then get to some best practices and tricks to make the situation a little less messy.

Dates can be represented in many ways—too many ways, actually. Even in English with the Roman calendar, you'll see many variants of a simple date:

- July 21 1987
- 21 Jul 1987
- 21/7/1987
- 7/21/1987

Among other problems, date representations can be ambiguous. In the previous examples, it's easy to determine that 7 stands for the month and 21 is the day of the month, because months don't go to 21. But how about `1/6/2012`? Is that referring to January 6 or June 1?

The month name varies by language within the Roman calendar. Even the year and month can have a different definition in other cultures.

Times have their own sources of grief, especially because of time zones and daylight savings time. If you look at a time zone map, the zones follow political and historic boundaries rather than crisp lines every 15 degrees (360 degrees / 24) of

longitude. And countries start and end daylight savings times on different days of the year. Southern hemisphere countries advance their clocks as their northern friends are winding theirs back, and vice versa.

Python's standard library has many date and time modules, including: `datetime`, `time`, `calendar`, `dateutil`, and others. There's some overlap, and it's a bit confusing.

Leap Year

Leap years are a special wrinkle in time. You probably know that every four years is a leap year (and the summer Olympics and the American presidential election). Did you also know that every 100 years is not a leap year, but that every 400 years is? Here's code to test various years for leapiness:

```
>>> import calendar
>>> calendar.isleap(1900)
False
>>> calendar.isleap(1996)
True
>>> calendar.isleap(1999)
False
>>> calendar.isleap(2000)
True
>>> calendar.isleap(2002)
False
>>> calendar.isleap(2004)
True
```

For the curious:

- A year has 365.242196 days (after one spin around the sun, the earth is about a quarter-turn on its axis from where it started).
- Add one day every four years. Now an average year has 365.242196 – 0.25 = 364.992196 days
- Subtract a day every 100 years. Now an average year has 364.992196 + 0.01 = 365.002196 days
- Add a day every 400 years. Now an average year has 365.002196 – 0.0025 = 364.999696 days

Close enough for now! We will not speak of leap seconds.

The datetime Module

The standard `datetime` module handles (which should not be a surprise) dates and times. It defines four main object classes, each with many methods:

- `date` for years, months, and days
- `time` for hours, minutes, seconds, and fractions
- `datetime` for dates and times together
- `timedelta` for date and/or time intervals

You can make a `date` object by specifying a year, month, and day. Those values are then available as attributes:

```
>>> from datetime import date
>>> halloween = date(2019, 10, 31)
>>> halloween
datetime.date(2019, 10, 31)
>>> halloween.day
31
>>> halloween.month
10
>>> halloween.year
2019
```

You can print a `date` with its `isoformat()` method:

```
>>> halloween.isoformat()
'2019-10-31'
```

The `iso` refers to ISO 8601, an international standard for representing dates and times. It goes from most general (year) to most specific (day). Because of this, it also sorts correctly: by year, then month, then day. I usually choose this format for date representation in programs, and for filenames that save data by date. The next section describes the more complex `strptime()` and `strftime()` methods for parsing and formatting dates.

This example uses the `today()` method to generate today's date:

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2019, 4, 5)
```

This one makes use of a `timedelta` object to add some time interval to a `date`:

```
>>> from datetime import timedelta
>>> one_day = timedelta(days=1)
>>> tomorrow = now + one_day
>>> tomorrow
datetime.date(2019, 4, 6)
>>> now + 17*one_day
datetime.date(2019, 4, 22)
>>> yesterday = now - one_day
>>> yesterday
datetime.date(2019, 4, 4)
```

The range of `date` is from `date.min` (year=1, month=1, day=1) to `date.max` (year=9999, month=12, day=31). As a result, you can't use it for historic or astronomical calculations.

The `datetime` module's `time` object is used to represent a time of day:

```
>>> from datetime import time
>>> noon = time(12, 0, 0)
>>> noon
datetime.time(12, 0)
>>> noon.hour
12
>>> noon.minute
0
>>> noon.second
0
>>> noon.microsecond
0
```

The arguments go from the largest time unit (hours) to the smallest (microseconds). If you don't provide all the arguments, `time` assumes all the rest are zero. By the way, just because you can store and retrieve microseconds doesn't mean you can retrieve time from your computer to the exact microsecond. The accuracy

of subsecond measurements depends on many factors in the hardware and operating system.

The `datetime` object includes both the date and time of day. You can create one directly, such as the one that follows, which is for January 2, 2019, at 3:04 A.M., plus 5 seconds and 6 microseconds:

```
>>> from datetime import datetime
>>> some_day = datetime(2019, 1, 2, 3, 4, 5, 6)
>>> some_day
datetime.datetime(2019, 1, 2, 3, 4, 5, 6)
```

The `datetime` object also has an `isoformat()` method:

```
>>> some_day.isoformat()
'2019-01-02T03:04:05.000006'
```

That middle `T` separates the date and time parts.

`datetime` has a `now()` method to return the current date and time:

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> now
datetime.datetime(2019, 4, 5, 19, 53, 7, 580562)
>>> now.year
2019
>>> now.month
4
>>> now.day
5
>>> now.hour
19
>>> now.minute
53
>>> now.second
7
>>> now.microsecond
580562
```

You can `combine()` a `date` object and a `time` object into a `datetime`:

```
>>> from datetime import datetime, time, date
>>> noon = time(12)
>>> this_day = date.today()
>>> noon_today = datetime.combine(this_day, noon)
>>> noon_today
datetime.datetime(2019, 4, 5, 12, 0)
```

You can yank the `date` and `time` from a `datetime` by using the `date()` and `time()` methods:

```
>>> noon_today.date()
datetime.date(2019, 4, 5)
>>> noon_today.time()
datetime.time(12, 0)
```

Using the time Module

It is confusing that Python has a `datetime` module with a `time` object, and a separate `time` module. Furthermore, the `time` module has a function called—wait for it—`time()`.

One way to represent an absolute time is to count the number of seconds since some starting point. *Unix time* uses the number of seconds since midnight on January 1, 1970.[1] This value is often called the *epoch*, and it is often the simplest way to exchange dates and times among systems.

The `time` module's `time()` function returns the current time as an epoch value:

```
>>> import time
>>> now = time.time()
>>> now
1554512132.778233
```

More than one billion seconds have ticked by since New Year's, 1970. Where did the time go?

You can convert an epoch value to a string by using `ctime()`:

```
>>> time.ctime(now)
'Fri Apr  5 19:55:32 2019'
```

In the next section, you'll see how to produce more attractive formats for dates and times.

Epoch values are a useful least-common denominator for date and time exchange with different systems, such as JavaScript. Sometimes, though, you need actual days, hours, and so forth, which `time` provides as `struct_time` objects. `localtime()` provides the time in your system's time zone, and `gmtime()` provides it in UTC:

```
>>> time.localtime(now)
time.struct_time(tm_year=2019, tm_mon=4, tm_mday=5, tm_hour=19,
tm_min=55, tm_sec=32, tm_wday=4, tm_yday=95, tm_isdst=1)
>>> time.gmtime(now)
time.struct_time(tm_year=2019, tm_mon=4, tm_mday=6, tm_hour=0,
tm_min=55, tm_sec=32, tm_wday=5, tm_yday=96, tm_isdst=0)
```

My `19:55` (Central time zone, Daylight Savings) was `00:55` in the next day in UTC (formerly called *Greenwich time* or *Zulu time*). If you omit the argument to `localtime()` or `gmtime()`, they assume the current time.

Some of the `tm_...` values in `struct_time` are a bit ambiguous, so take a look at Table 13-1 for more details.

Table 13-1. `struct_time` values

Index	Name	Meaning	Values
0	`tm_year`	Year	`0000` to `9999`
1	`tm_mon`	Month	`1` to `12`
2	`tm_mday`	Day of month	`1` to `31`
3	`tm_hour`	Hour	`0` to `23`
4	`tm_min`	Minute	`0` to `59`
5	`tm_sec`	Second	`0` to `61`
6	`tm_wday`	Day of week	`0` (Monday) to `6` (Sunday)
7	`tm_yday`	Day of year	`1` to `366`
8	`tm_isdst`	Daylight savings?	`0` = no, `1` = yes, `-1` = unknown

If you don't want to type all those `tm_...` names, `struct_time` also acts like a named tuple (see "Named Tuples"), so you can use the indexes from the previous table:

```
>>> import time
>>> now = time.localtime()
>>> now
time.struct_time(tm_year=2019, tm_mon=6, tm_mday=23, tm_hour=12,
tm_min=12, tm_sec=24, tm_wday=6, tm_yday=174, tm_isdst=1)
>>> now[0]
2019
print(list(now[x] for x in range(9)))
[2019, 6, 23, 12, 12, 24, 6, 174, 1]
```

`mktime()` goes in the other direction, converting a `struct_time` object to epoch seconds:

```
>>> tm = time.localtime(now)
>>> time.mktime(tm)
1554512132.0
```

This doesn't exactly match our earlier epoch value of `now()` because the `struct_time` object preserves time only to the second.

NOTE

Some advice: wherever possible, *use UTC* instead of time zones. UTC is an absolute time, independent of time zones. If you have a server, *set its time to UTC*; do not use local time.

More advice: *never use daylight savings time* if you can avoid it. If you use daylight savings time, an hour disappears at one time of year ("spring ahead") and occurs twice at another time ("fall back"). For some reason, many organizations use local time with daylight savings in their computer systems, but are mystified twice every year by that spooky hour.

Read and Write Dates and Times

`isoformat()` is not the only way to write dates and times. You already saw the `ctime()` function in the `time` module, which you can use to convert epochs to strings:

```
>>> import time
>>> now = time.time()
>>> time.ctime(now)
'Fri Apr  5 19:58:23 2019'
```

You can also convert dates and times to strings by using `strftime()`. This is provided as a method in the `datetime`, `date`, and `time` objects, and as a function in the `time` module. `strftime()` uses format strings to specify the output, which you can see in Table 13-2.

Table 13-2. Output specifiers for `strftime()`

Format string	Date/time unit	Range
`%Y`	year	`1900`-...
`%m`	month	`01`-`12`
`%B`	month name	`January`, ...
`%b`	month abbrev	`Jan`, ...
`%d`	day of month	`01`-`31`
`%A`	weekday name	`Sunday`, ...
`a`	weekday abbrev	`Sun`, ...
`%H`	hour (24 hr)	`00`-`23`
`%I`	hour (12 hr)	`01`-`12`
`%p`	AM/PM	`AM`, `PM`
`%M`	minute	`00`-`59`
`%S`	second	`00`-`59`

Numbers are zero-padded on the left.

Here's the `strftime()` function provided by the `time` module. It converts a `struct_time` object to a string. We'll first define the format string `fmt` and use it again later:

```
>>> import time
>>> fmt = "It's %A, %B %d, %Y, local time %I:%M:%S%p"
>>> t = time.localtime()
>>> t
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=13, tm_hour=15,
tm_min=23, tm_sec=46, tm_wday=2, tm_yday=72, tm_isdst=1)
```

```
>>> time.strftime(fmt, t)
"It's Wednesday, March 13, 2019, local time 03:23:46PM"
```

If we try this with a `date` object, only the date parts will work, and the time defaults to midnight:

```
>>> from datetime import date
>>> some_day = date(2019, 7, 4)
>>> fmt = "It's %A, %B %d, %Y, local time %I:%M:%S%p"
>>> some_day.strftime(fmt)
"It's Thursday, July 04, 2019, local time 12:00:00AM"
```

For a `time` object, only the time parts are converted:

```
>>> from datetime import time
>>> fmt = "It's %A, %B %d, %Y, local time %I:%M:%S%p"
>>> some_time = time(10, 35)
>>> some_time.strftime(fmt)
"It's Monday, January 01, 1900, local time 10:35:00AM"
```

You won't want to use the day parts from a `time` object, because they're meaningless.

To go the other way and convert a string to a date or time, use `strptime()` with the same format string. There's no regular expression pattern matching; the non-format parts of the string (without `%`) need to match exactly. Let's specify a format that matches *year-month-day*, such as `2019-01-29`. What happens if the date string you want to parse has spaces instead of dashes?

```
>>> import time
>>> fmt = "%Y-%m-%d"
>>> time.strptime("2019 01 29", fmt)
Traceback (most recent call last):
  File "<stdin>",
    line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/_strptime.py",
    line 571, in _strptime_time
    tt = _strptime(data_string, format)[0]
  File "/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/_strptime.py",
    line 359, in _strptime(data_string, format))
ValueError: time data '2019 01 29' does not match format '%Y-%m-%d
```

If we feed `strptime()` some dashes, is it happy now?

```
>>> import time
>>> fmt = "%Y-%m-%d"
>>> time.strptime("2019-01-29", fmt)
time.struct_time(tm_year=2019, tm_mon=1, tm_mday=29, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=1, tm_yday=29, tm_isdst=-1)
```

Or fix the `fmt` string to match the date string:

```
>>> import time
>>> fmt = "%Y %m %d"
>>> time.strptime("2019 01 29", fmt)
time.struct_time(tm_year=2019, tm_mon=1, tm_mday=29, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=1, tm_yday=29, tm_isdst=-1)
```

Even if the string seems to match its format, an exception is raised if a value is out of range (file names truncated for space):

```
>>> time.strptime("2019-13-29", fmt)
Traceback (most recent call last):
  File "<stdin>",
    line 1, in <module>
  File ".../3.7/lib/python3.7/_strptime.py",
    line 571, in _strptime_time
    tt = _strptime(data_string, format)[0]
  File ".../3.7/lib/python3.7/_strptime.py",
    line 359, in _strptime(data_string, format))
ValueError: time data '2019-13-29' does not match format '%Y-%m-%d
```

Names are specific to your *locale*—internationalization settings for your operating system. If you need to print different month and day names, change your locale by using `setlocale()`; its first argument is `locale.LC_TIME` for dates and times, and the second is a string combining the language and country abbreviation. Let's invite some international friends to a Halloween party. We'll print the month, day, and day of week in US English, French, German, Spanish, and Icelandic (Icelanders have real elves):

```
>>> import locale
>>> from datetime import date
```

```
>>> halloween = date(2019, 10, 31)
>>> for lang_country in ['en_us', 'fr_fr', 'de_de', 'es_es', 'is_is',]:
...     locale.setlocale(locale.LC_TIME, lang_country)
...     halloween.strftime('%A, %B %d')
...
'en_us'
'Thursday, October 31'
'fr_fr'
'Jeudi, octobre 31'
'de_de'
'Donnerstag, Oktober 31'
'es_es'
'jueves, octubre 31'
'is_is'
'fimmtudagur, október 31'
>>>
```

Where do you find these magic values for `lang_country`? This is a bit wonky, but you can try this to get all of them (there are a few hundred):

```
>>> import locale
>>> names = locale.locale_alias.keys()
```

From `names`, let's get just locale names that seem to work with `setlocale()`, such as the ones we used in the preceding example—a two-character language code followed by an underscore and a two-character country code:

```
>>> good_names = [name for name in names if \
len(name) == 5 and name[2] == '_']
```

What do the first five look like?

```
>>> good_names[:5]
['sr_cs', 'de_at', 'nl_nl', 'es_ni', 'sp_yu']
```

So, if you wanted all the German language locales, try this:

```
>>> de = [name for name in good_names if name.startswith('de')]
>>> de
['de_at', 'de_de', 'de_ch', 'de_lu', 'de_be']
```

NOTE

If you run `set_locale()` and get the error

```
locale.Error: unsupported locale setting
```

that locale is not supported by your operating system. You'll need to figure out what your operating system needs to add it. This can happen even if Python told you (using `locale.locale_alias.keys()`) that it was a good locale. I had this error when testing on macOS with the locale `cy_gb` (Welsh, Great Britain), even though it had accepted `is_is` (Icelandic) in the preceding example.

All the Conversions

Figure 13-1 (from the Python wiki) summarizes all the standard Python time interconversions.

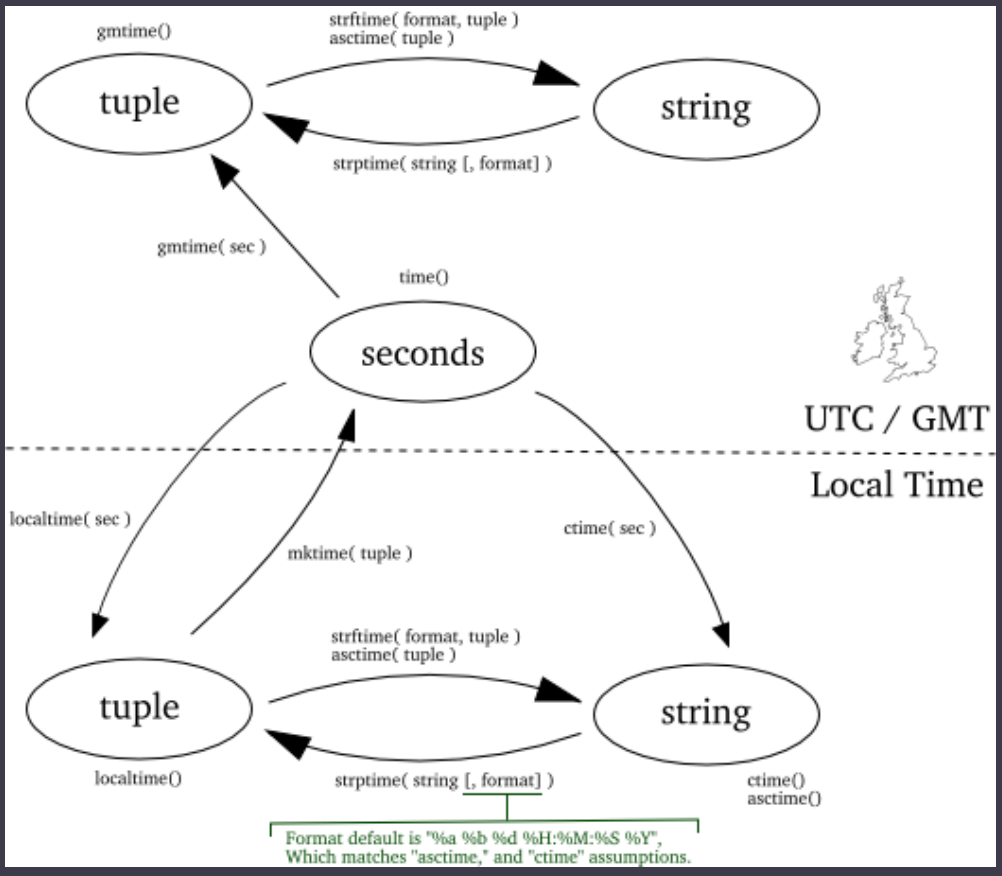

Figure 13-1. Date and time conversions

Alternative Modules

If you find the standard library modules confusing, or lacking a particular conversion that you want, there are many third-party alternatives. Here are just a few of them:

`arrow`
: Combines many date and time functions with a simple API

`dateutil`
: Parses almost any date format and handles relative dates and times well

`iso8601`
: Fills in gaps in the standard library for the ISO8601 format

`fleming`
: Many time zone functions

`maya`
: Intuitive interface to dates, times, and intervals

`dateinfer`
: Guesses the right format strings from date/time strings

Coming Up

Files and directories need love, too.

Things to Do

13.1 Write the current date as a string to the text file *today.txt*.

13.2 Read the text file *today.txt* into the string `today_string`.

13.3 Parse the date from `today_string`.

13.4 Create a date object of your day of birth.

13.5 What day of the week was your day of birth?

13.6 When will you be (or when were you) 10,000 days old?

1 This starting point is roughly when Unix was born, ignoring those pesky leap seconds.