5 Logs & Time

Key terms: `date` `timedelta` `datetime`[13] `dateutil` `calendar`

Reading: Basic date and time types[14]

Exercise: Write a program that demonstrates a generator yielding the number of seconds after midnight when each access occurred, from the beginning of `/etc/httpd/logs/access_log`.

The Gregorian calendar is complicated. The solar year isn't an integer multiple of solar days (it's about 365.24) nor lunar cycles (about 12.39). Months have 28 to 31 days; under Daylight Savings schemes, days have 23 to 25 hours; and leap seconds make some minutes last 61 seconds. Then there are time zones, whose borders aren't straight lines. All these elements complicate timing, so thankfully Python includes infrastructure to handle the complexity.

Working with timestamps can be tricky if you run into other ancient traditions of timestamping such as the epoch timestamp or the `time.struct_time` model. Python can work with these, but you'll hope not to, because they present complications compared to the native types, such as requiring conversion.

There are various different string representations of the same date, such as the POSIX style offered by `datetime.timestamp()` and the various locale variations expected around the world. When time zone information is present, a timestamp is "aware", as opposed to "naive". Any kind of string representations of dates can be loaded once the format is specified using `datetime.strptime()`. The most common zone to use other than the local one is, of course, Coordinated Universal Time, UTC.

In the `datetime` package, `date` and `time` objects are just as they sound: idealized measures devoid of geography and politics. A `datetime` combines them into a single point in time — subject to assumptions about all those complex details. Time types are immutable. Make new times as needed by adding or subtracting a `timedelta`; these can be defined directly (`datetime.timedelta(days=1)`) or made from the differences between two `datetime`s.

```
lifetime = date(1852,11,27) - date(1815,12,10)
print("The Countess of Lovelace lived {} days.".format(lifetime.days))
```

Loops over time periods can also support chronological analysis and presentation. Here's a way to print out any year's calendar by weeks.

```
# Produce a weekly calendar of the current year:

import datetime
thisYear = datetime.date.today().year
today = datetime.date(thisYear,1,1)

# Loop over the days in the year, splitting them into weeks.

print ( '   ' * ( (1+today.weekday()) % 7), end='' )
while today < datetime.date(thisYear+1,1,1):
 ending = '\n' if today.weekday() == 5 else ' '
 print ( '{:2}'.format(today.day), end=ending )
 today += datetime.timedelta(1)
print()
```

Next, we will look into profiling the time performance of our code.

[13] https://docs.python.org/3/library/datetime.html
[14] https://docs.python.org/3/library/datetime.html