pytz: The Fastest Footgun in the West

Paul Ganssle

Published Mon 19 March 2018 *(updated Wed 01 April 2020)* in programming

python timezones pytz dateutil datetime

Whenever I give a talk about time zones, someone comes up to me afterwards and tells me that they have broken code *currently in production*, because they misunderstood how `pytz` works. This is because `pytz` uses its own non-standard interface for handling time zone information that is partially but not entirely compatible with the way Python's `datetime` library was intended to work, which leads to a lot of confusion from people naively using `pytz` as a time zone provider. This incompatibility is why, as of Python 3.6, the tzinfo documentation recommends `dateutil.tz` rather than `pytz` as an IANA time zone provider. [1]

In this post, I will cover both time zone models and if I cannot convince you to switch to `dateutil.tz`, at least provide some intuition about the differences between `pytz` and the standard time zone model.

Python's time zone model

In the `datetime` module, Python provides support for *time zones* rather than time zone offsets - which is to say that a `datetime.tzinfo` object is expected to provide not a fixed offset and name but a set of rules for what the time zone information is *as a function of the datetime*. This is so that something like this will work:

```
from dateutil import tz
from datetime import datetime, timedelta

NYC = tz.gettz('America/New_York')
dt_winter = datetime(2018, 2, 14, 12, tzinfo=NYC)
print(dt_winter)
# 2018-02-14 12:00:00-05:00

dt_spring = dt_winter + timedelta(days=60)
print(dt_spring)
# 2018-04-15 12:00:00-04:00
```

If `NYC` were a static, fixed offset, you'd need to attach a *different* `tzinfo` to each datetime depending on whether or not you're in standard or daylight time, and any time you did any math on your datetime, you'd have to redo the calculations in case the offset had changed. Thus, Python's model is that any `tzinfo` subclass should implement the following three methods:

- `tzname(self, dt)` : The name of the offset at the given datetime (e.g. EST, PDT)
- `utcoffset(self, dt)` : The offset from UTC at the given datetime
- `dst(self, dt)` : The difference between the current offset and the zone's "standard offset" [2]

These values are not only implemented as a function, they are also invoked *lazily*, so there are no hooks in the `datetime` constructor that call these – they are only invoked when a user wants to know one or more of these pieces of information.

pytz's time zone model

The biggest mistake people make with `pytz` is simply attaching its time zones to the constructor, since that is the standard way to add a time zone to a `datetime` in Python. If you try and do that, the best case scenario is that you'll get something obviously absurd:

```
import pytz
from datetime import datetime, timedelta

NYC = pytz.timezone('America/New_York')
dt = datetime(2018, 2, 14, 12, tzinfo=NYC)
print(dt)
# 2018-02-14 12:00:00-04:56
```

Why is the time offset -04:56 and not -05:00? Because that was the local solar mean time in New York before standardized time zones were adopted, and is thus the first entry in the `America/New_York` time zone. Why did `pytz` return that? Because unlike the standard library's model of lazily-computed time zone information, `pytz` takes an eager calculation approach. Whenever you construct an aware `datetime` from a naive one, you need to call the `localize` function on it:

```
dt = NYC.localize(datetime(2018, 2, 14, 12))
print(dt)
# 2018-02-14 12:00:00-05:00
```

Each `pytz` time zone contains a list of possible fixed offset "time zone" objects that are valid at different times in that zone, and the `localize` function figures out which one is valid at that local date and time and attaches it. In this case it detects correctly that 2018-02-14 should be EST with offset -05:00 and DST -00:00. Now what happens when you perform the arithmetic on a localized datetime?

```
from datetime import timedelta

dt_spring = dt + timedelta(days=60)
print(dt_spring)
# 2018-04-15 12:00:00-05:00
```

Since the `localize` function eagerly attached EST to the datetime, the offset is not updated in response to the arithmetic. In order to fix this error, any time you do any arithmetic on a `pytz`-aware datetime, you need to call the `normalize` function:

```
print(NYC.normalize(dt_spring))
# 2018-04-15 13:00:00-04:00
```

This is, again, just eagerly doing the calculation that would be done lazily by a `dateutil.tz` time zone object [3].

Ambiguous datetimes

Why was `pytz` designed this way, given that it doesn't mesh well with Python's standard time zone model? Consider the scenario of an *ambiguous datetime*, which occurs during a daylight saving time transition, e.g. `2018-11-04 01:30-04:00`, and an hour later, `2018-11-04 01:30-05:00`. How would you write a function that takes the `2018-11-04 01:30` portion of that datetime, and returns the correct answer? You can't, because there are *two* correct answers.

`pytz` is able to solve this problem because during the `localize` step, the time zone could take in the additional information of whether you wanted to be on the DST or STD side of the transition:

```
dt_dst = NYC.localize(datetime(2018, 11, 4, 1, 30), is_
print(dt_dst)
# 2018-11-04 01:30:00-04:00

dt_std = NYC.localize(datetime(2018, 11, 4, 1, 30), is_
print(dt_std)
# 2018-11-04 01:30:00-05:00
```

This is because some of the information about the `datetime` (which side of DST it represents) is now encoded in the `tzinfo` that was attached to it. Using the standard Python interface, this problem was not solved until Python 3.6 with the introduction of PEP 495, which added the `fold` attribute to the `datetime` class. This allows the "which side of an ambiguous datetime do I fall on" decision to be encoded *in the datetime itself*, allowing for lazy calculation of ambiguous datetimes. If `dt.fold` is 0, ambiguous datetimes resolve to the *first* occurrence of a time in a given zone, if it's 1, they resolve to the *second* occurrence. So to represent the example above:

```
from dateutil import tz
NYC_du = tz.gettz('America/New_York')

dt_dst = datetime(2018, 11, 4, 1, 30, fold=0, tzinfo=NY
print(dt_dst)
# 2018-11-04 01:30:00-04:00

dt_std = datetime(2018, 11, 4, 1, 30, fold=1, tzinfo=NY
print(dt_std)
# 2018-11-04 01:30:00-05:00
```

Additionally, `dateutil` is able to backport this to earlier versions of Python by providing a `tz.enfold` function that will, if necessary, create a `datetime` subclass providing the `fold` attribute. So, on Python 2.7 you get:

```
>>> tz.enfold(datetime(2018, 11, 4, 1, 30), fold=0)
datetime.datetime(2018, 11, 4, 1, 30)

>>> tz.enfold(datetime(2018, 11, 4, 1, 30), fold=1)
_DatetimeWithFold(2018, 11, 4, 1, 30)
```

Now that this issue is resolved, `pytz` no longer has the advantage of being the best way to handle ambiguous datetimes, but retains the *disadvantage* of its somewhat clunky interface and eagerly-calculated time zone information.

Fastest footgun in the west

The title of this post claims that `pytz` is the *fastest* footgun in the west, by which I mean that `pytz` is a quite well-optimized library, and historically it has been faster than `dateutil.tz`. The performance gap has been significantly reduced in the last several releases [4], but there is still a persistent gap in performance for some use cases owing to the lazy vs. eager nature of the calculations. To demonstrate, I've timed `pytz==2018.3` and `python-dateutil==2.7.0` on Python 3.6, using IPython's `%timeit` magic, and put the timing numbers in comments after each function:[6]

```
import pytz
from dateutil import tz

from datetime import datetime

NYC_p = pytz.timezone('America/New_York')    # 1.53 µs
NYC_d = tz.gettz('America/New_York')         # 863 ns

dt_p = NYC_p.localize(datetime(2018, 11, 1))     # 35.4 
dt_d = datetime(2018, 11, 1, tzinfo=NYC_d)       # 1.38 

dt_p.utcoffset()         # 655 ns
dt_d.utcoffset()         # 13.9 µs
```

As you can see, `pytz` time zones are more costly to construct and the initial `localize` call is much slower than even `dateutil`'s `utcoffset()` call, but `pytz`'s `utcoffset()` call beats out `dateutil`'s by a wide margin (because the result is cached). If you plan to construct a bunch of datetimes and query their time zone information relatively infrequently (an average of 2-3 times per datetime), it seems that `dateutil` beats `pytz` in "total time to first `utcoffset` call":

```
NYC_p.localize(datetime(2018, 11, 1)).utcoffset()    # 3
datetime(2018, 11, 1, tzinfo=NYC_d).utcoffset()      # 1
```

The story is similar for converting from one time zone to another (though `pytz` does better than I would expect):

```
LA_p = pytz.timezone('America/Los_Angeles')
LA_d = tz.gettz('America/Los_Angeles')

dt_p.astimezone(LA_p)       # 7.53 µs
dt_d.astimezone(LA_d)       # 31.5 µs
```

`dateutil` is slower in this operation because it needs to calculate the UTC offset for both New York and Los Angeles, and `pytz` evidently is faster at creating localized datetimes from UTC than from naive datetimes (otherwise I would expect to pay at least the cost of another `localize`). If you are planning on doing only one time zone conversion per localized `datetime`, though, the full cost of the operation is:

```
NYC_p.localize(datetime(2018, 11, 1)).astimezone(LA_p)
datetime(2018, 11, 1, tzinfo=NYC_d).astimzone(LA_d)
```

In neither case is there a clear "overall" winner on performance. If you plan on localizing a datetime and then repeatedly query its `utcoffset` or convert it to other time zones, `pytz` may perform better, because the offset values are cached. [5] If you are only making an average of 2-3 `utcoffset` calls per `datetime`, you may get better performance out of `dateutil`.

I suspect that for most practical cases, the marginal cost of additional time zone calculations can be narrowed significantly with little memory overhead by a implementing even a modest Least Recently Used cache *(update: not as feasible as I originally thought)* [7], but it is worth noting that `pytz`'s design brings an effectively infinite cache "for free". In any case, the standard disclaimer applies — don't worry about these kinds of micro-benchmarks until you're certain that they are the bottleneck of your operation.

Conclusions

At the time of its creation, `pytz` was cleverly designed to optimize for performance and correctness, but with the changes introduced by PEP 495 and the performance improvements to `dateutil`, the reasons to use it are dwindling. As mentioned in the previous sections, there are some use cases where `pytz`'s IANA time zones are faster, but common use cases where they are slower as well. Historically, they provided a "more correct" time zone implementation, but now they solve the ambiguous time problem in a way that is inconsistent with Python's time zone model.

The biggest reason to use `dateutil` over `pytz` is the fact that `dateutil` uses the standard interface and `pytz` doesn't, and as a result it is very easy to `pytz` incorrectly. Even if *you* now know the right way to use `pytz`, are you sure that you're not going to pass your datetime to a function expecting something that uses the standard `tzinfo` interface? Are you sure that anyone maintaining your code or consuming its outputs are going to know to avoid these mistakes?

Footnotes

[1] Scroll down to the "See also" section, which doesn't have its own anchor link.

[2] I'll note that the existence of a "standard offset" may be an unwarranted assumption of Python's time zone model. There is some identifiable "standard" offset in all time zones that I know

about, but there's nothing stopping people from observing a strange time zone that switches between 3 equally valid "time zones" during the course of a year for reasons *other* than daylight saving time.

[3] By providing a completely different `tzinfo` object, this also interferes with Python's model for time zone aware arithmetic semantics, which depends on two datetimes in the same zone satisfying `dt1.tzinfo is dt2.tzinfo`.

[4] Among other improvements in version 2.7.0, `dateutil` has added a `dateutil.tz.UTC` singleton (previously `dateutil.tz.tzutc()` constructed a new object for each call) and started caching calls to `tz.gettz` and generally reduced the number of constructor calls to other time zone objects.

[5] It is possible, of course, to simply add caching to `dateutil`'s time zone functions, but `pytz` has the inherent advantage that the `tzinfo` cache is built in to each `datetime` object – in order to implement a cache for `dateutil`'s time zones, each `tzinfo` has to maintain a history of `datetime` values for which the offset has been calculated and map that to the proper lookup values. By contrast, `pytz` stores this mapping in the `tzinfo` argument, since each `dateutil` stores a reference to the offset that it was localized to, at the (memory) cost of proliferating one extra `tzinfo` object per possible offset (in most zones this will be 3-5 extra `tzinfo` objects).

[6] Keep in mind, both `dateutil` and `pytz` make extensive use of caching, so these numbers are the "asymptotic" behavior. These numbers are mainly useful if you end up doing basically the same thing over and over again in a tight loop.

[7] **Update 2020-04-01**: An LRU cache for datetimes is not as feasible as I had originally thought, because the UTC offset is a part of the `datetime`'s hash calculation, so it is not possible to simply look it up in a dictionary without the expensive operation of constructing a new `datetime`. There are other strategies and cache designs that could work, and I proposed one such scheme in bpo-35723, but likely the need for these will be mooted by the acceptance of PEP 615, since every operation in the C extension in the reference implementation is considerably faster than *pytz*, even without any caching.