Working With Time Zones

Everything You Wish You Didn't Need to K
(`zoneinfo` Edition)

Paul Ganssle

@pganssle

This talk on Github: pganssle-talks/pycon-us-2023-timezones

@pganssle
ganssle.io

Introduction

UTC

- Reference time zone
- Monotonic-ish (what's a few leap seconds between friends?)

@pganssle
ganssle.io

Introduction

UTC

- Reference time zone
- Monotonic-ish (what's a few leap seconds between friends?)

Time zones vs. Offsets

- `UTC-6` is an offset
- `America/Chicago` is a time zone
- `CST` is a highly context-dependent abbreviation:
  - Central Standard Time (`UTC-6`)
  - Cuba Standard Time (`UTC-5`)
  - China Standard Time (`UTC+8`)

@pganssle
ganssle.io

Complicated time zones

Non-integer offsets

- `Australia/Adelaide` (+09:30)
- `Asia/Kathmandu` (+05:45)
- `Africa/Monrovia` (+00:44:30) (Before 1979)

@pganssle
ganssle.io

Complicated time zones

Non-integer offsets

- `Australia/Adelaide` (+09:30)
- `Asia/Kathmandu` (+05:45)
- `Africa/Monrovia` (+00:44:30) (Before 1979)

Change of DST status without offset change

- Portugal, 1992
  - `WET (+0 STD) → WEST (+1 DST) 1992-03-29`
  - `WEST (+1 DST) → CET (+1 STD) 1992-09-27`
- Portugal, 1996
  - `CET (+1 STD) -> WEST (+1 DST) 1996-03-31`
  - `WEST (+1 DST) -> WET (+0 STD) 1996-10-27`

Complicated time zones

More than one DST transition per year

- Morroco, 2012
  - `WET (+0 STD) -> WEST (+1 DST) 2012-04-29`
  - `WEST (+1 DST) -> WET (+0 STD) 2012-07-20`
  - `WET (+0 STD) -> WEST (+1 DST) 2012-08-20`
  - `WEST (+1 DST) -> WET (+0 STD) 2012-09-30`

... and Morocco in 2013-present, and Egypt in 2010 and 2014, and Palestine in 20

@pganssle
ganssle.io

Complicated time zones

Missing days

- Christmas Island (Kiritimati), December 31, 1994 (UTC-10 → UTC+14)

```
>>> dt_before = datetime(1994, 12, 30, 23, 59, tzinfo=ZoneInfo('Pacific/Kiritimati'))
>>> dt_after = add_absolute(dt_before, timedelta(minutes=2))

>>> print(dt_before)
1994-12-30 23:59:00-10:00

>>> print(dt_after)
1995-01-01 00:01:00+14:00
```

Also Samoa on January 29, 2011.

@pganssle
ganssle.io

Complicated time zones

Double days

- Kwajalein Atoll, 1969

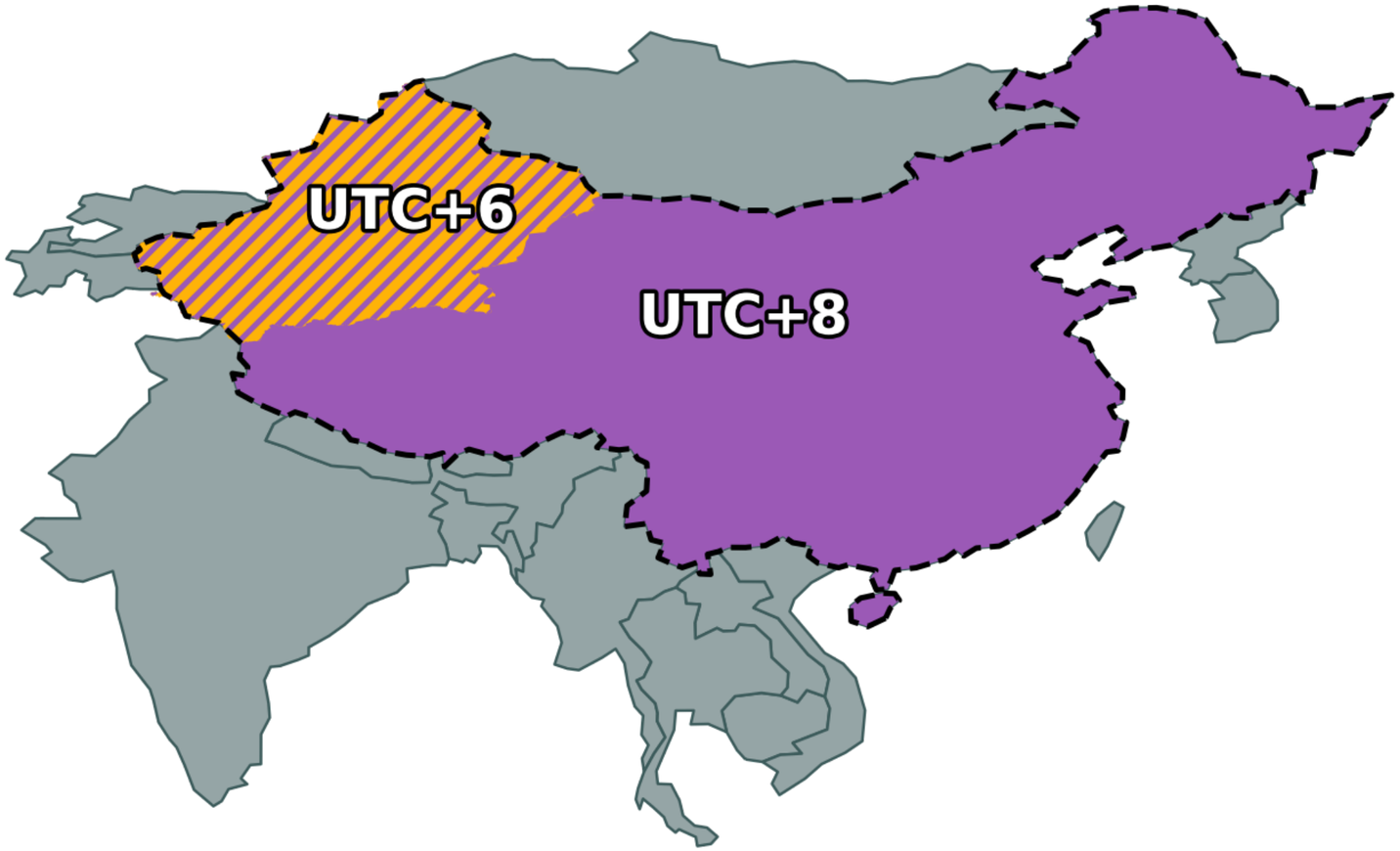
```
>>> dt_before = datetime(1969, 9, 30, 11, 59, tzinfo=ZoneInfo('Pacific/Kwajalein'))
>>> dt_after = add_absolute(dt_before, timedelta(minutes=2))

>>> print(dt_before)
1969-09-30 11:59:00+11:00

>>> print(dt_after)
1969-09-30 12:01:00+11:00
```

@pganssle
ganssle.io

Asia/Shanghai

@pganssle
ganssle.io

Asia/Urumqi



@pganssle
ganssle.io

Why do we need to work with time zones at all?

```python
from dateutil import rrule as rr
from datetime import datetime, timezone
from zoneinfo import ZoneInfo

# Close of business in New York on weekdays
closing_times = rr.rrule(freq=rr.DAILY, byweekday=(rr.MO, rr.TU, rr.WE, rr.TH, rr.FR),
                         byhour=17, dtstart=datetime(2023, 3, 8, 17), count=5)

NYC = ZoneInfo("America/New_York")
for dt in closing_times:
    print(dt.replace(tzinfo=NYC))
```

```
2023-03-08 17:00:00-05:00
2023-03-09 17:00:00-05:00
2023-03-10 17:00:00-05:00
2023-03-13 17:00:00-04:00
2023-03-14 17:00:00-04:00
```

```python
# Get close of business in UTC
for dt in closing_times:
    print(dt.replace(tzinfo=NYC).astimezone(timezone.utc))
```

```
2023-03-08 22:00:00+00:00
2023-03-09 22:00:00+00:00
2023-03-10 22:00:00+00:00
2023-03-13 21:00:00+00:00
2023-03-14 21:00:00+00:00
```

@pganssle
ganssle.io

When storing datetimes where the *wall time* matters (e.g. meetin
store local time, because the mapping between UTC and local time

Python's Time Zone Model: `tzinfo`

- Time zones are provided by *subclassing* `tzinfo`.
- Information provided is a function of the datetime:
  - `tzname`: The (usually abbreviated) name of the time zone at the given datetime
  - `utcoffset`: The offset from UTC at the given datetime
  - `dst`: The size of the `datetime`'s DST offset (usually 0 or 1 hour)

@pganssle
ganssle.io

Python's Time Zone Model: `tzinfo`

- Time zones are provided by *subclassing* `tzinfo`.
- Information provided is a function of the datetime:
  - `tzname`: The (usually abbreviated) name of the time zone at the given datetime
  - `utcoffset`: The offset from UTC at the given datetime
  - ~~`dst`: The size of the `datetime`'s DST offset (usually 0 or 1 hour)~~

@pganssle
ganssle.io

History of Python's Time Zones

When `datetime` was introduced in Python 2.3, there were *no* concrete time zones in the st

```
from dateutil import relativedelta as rd  # Cheating...

class ET(tzinfo):
    def utcoffset(self, dt):
        if self.isdaylight(dt):
            return timedelta(hours=-4)
        else:
            return timedelta(hours=-5)

    def dst(self, dt):
        if self.isdaylight(dt):
            return timedelta(hours=1)
        else:
            return timedelta(hours=0)

    def tzname(self, dt):
        return "EDT" if self.isdaylight(dt) else "EST"

    def isdaylight(self, dt):
        dst_start = datetime(dt.year, 1, 1) + rd.relativedelta(month=3, weekday=rd.SU(+2),
                                                               hour=2)
        dst_end = datetime(dt.year, 1, 1) + rd.relativedelta(month=11, weekday=rd.SU,
                                                             hour=2)

        return dst_start <= dt.replace(tzinfo=None) < dst_end
```

@pganssle
ganssle.io

History of Python's Time Zones: Concrete Time Zon

- UTC / Fixed Offsets
- Local time
- IANA Time Zones

@pganssle
ganssle.io

History of Python's Time Zones: Concrete Time Zon

- UTC / Fixed Offsets ✔ Added in 3.2
- Local time
- IANA Time Zones

What's New In Python 3.2

datetime and time

- The `datetime` module has a new type `timezone` that implements the `tzinfo` interface by returning a fixed UTC offset and timezone name. This makes it easier to create timezone-aware datetime objects:

```
>>> from datetime import datetime, timezone

>>> datetime.now(timezone.utc)
datetime.datetime(2010, 12, 8, 21, 4, 2, 923754, tzinfo=datetime.timezone.utc)

>>> datetime.strptime("01/01/2000 12:00 +0000", "%m/%d/%Y %H:%M %z")
datetime.datetime(2000, 1, 1, 12, 0, tzinfo=datetime.timezone.utc)
```

@pganssle
ganssle.io

Ambiguous times

Ambiguous times are times where the same "wall time" occurs twice, such as during a DST to

```
from dateutil import tz

dt1 = datetime(2004, 10, 31, 4, 30, tzinfo=timezone.utc)
for i in range(4):
    dt = (dt1 + timedelta(hours=i)).astimezone(NYC)
    print('{} | {} |  {}'.format(dt, dt.tzname(),
                                 'Ambiguous' if tz.datetime_ambiguous(dt)
                                 else 'Unambiguous'))
```

```
2004-10-31 00:30:00-04:00 | EDT |  Unambiguous
2004-10-31 01:30:00-04:00 | EDT |  Ambiguous
2004-10-31 01:30:00-05:00 | EST |  Ambiguous
2004-10-31 02:30:00-05:00 | EST |  Unambiguous
```

There can be multiple times in a time zone differentiated by their offset!

@pganssle
ganssle.io

Imaginary times

The complement of ambiguous times is imaginary times — wall times that don't exist in a given time z
STD to DST transition.

```
dt1 = datetime(2004, 4, 4, 6, 30, tzinfo=timezone.utc)
for i in range(3):
    dt = (dt1 + timedelta(hours=i)).astimezone(NYC)
    print(f'{dt} | {dt.tzname()} ')
```

```
2004-04-04 01:30:00-05:00 | EST
2004-04-04 03:30:00-04:00 | EDT
2004-04-04 04:30:00-04:00 | EDT
```

Notice the lack of a `2004-04-04 02:30:00`!

@pganssle
ganssle.io

`pytz`'s time zone model

- `tzinfo` is attached *by the time zone object itself*:

```
>>> LOS_p = pytz.timezone('America/Los_Angeles')
>>> dt = LOS_p.localize(datetime(2017, 8, 11, 14, 0))
>>> print_tzinfo(dt)


2017-08-11 14:00:00-0700
    tzname:   PDT;      UTC Offset:  -7.00h;        DST:      1.0h
```

- `tzinfo`s are all *static offsets*:

```
>>> print(repr(LOS_p))
<DstTzInfo 'America/Los_Angeles' LMT-1 day, 16:07:00 STD>

>>> print(repr(dt.tzinfo))
<DstTzInfo 'America/Los_Angeles' EDT-1 day, 20:00:00 DST>
```

- Python's model is designed to be lazy, but `pytz`'s model is *eager*

@pganssle
ganssle.io

Handling ambiguous and imaginary times in pyt

Because offsets are eagerly evaluated, it is possible to represent `datetimes` that differ only in thei
different offsets to them. `pytz`'s `is_dst` has three modes:

- `is_dst=False` (default): choose the STD side if ambiguous

```
>>> ambiguous = datetime(2004, 10, 31, 1, 30)

>>> print_tzinfo(LOS_p.localize(ambiguous, is_dst=False))
2004-10-31 01:30:00-0500
    tzname:   EST;      UTC Offset:  -5.00h;        DST:      0.0h
```

- `is_dst=True`: choose the DST side if ambiguous

```
>>> print_tzinfo(LOS_p.localize(ambiguous, is_dst=True))
2004-10-31 01:30:00-0400
    tzname:   EDT;      UTC Offset:  -4.00h;        DST:      1.0h
```

- `is_dst=None`: Throw an error if ambiguous

```
>>> LOS_p.localize(ambiguous, is_dst=None)
AmbiguousTimeError                          Traceback (most recent call last)
...
    362         if is_dst is None:
--> 363             raise AmbiguousTimeError(dt)

AmbiguousTimeError: 2004-10-31 01:30:00
```

@pganssle
ganssle.io

Problems with `pytz`'s time zone model

- Requires eager calculation — directly attaching a `pytz` timezone gives the wrong results:

```
>>> dt = datetime(2020, 5, 1, tzinfo=LOS_p)
>>> print_tzinfo(dt)
2020-05-01 00:00:00-0753
    tzname:   LMT;      UTC Offset:  -7.88h;        DST:      0.0h
```

- You must `normalize()` datetimes after you've done some arithmetic on them:

```
>>> dt = LOS_p.localize(datetime(2020, 5, 1))
>>> dt_add = dt + timedelta(days=180)

>>> print_tzinfo(dt_add)
2018-02-07 14:00:00-0700
    tzname:   PDT;      UTC Offset:  -7.00h;        DST:      1.0h

>>> print_tzinfo(LOS_p.normalize(dt_add))
2018-02-07 13:00:00-0800
    tzname:   PST;      UTC Offset:  -8.00h;        DST:      0.0h
```

@pganssle
ganssle.io

PEP 495: Local Time Disambiguation

- First introduced in Python 3.6 to solve the ambiguous time problem
- Introduces the `fold` attribute of `datetime`
- Changes to aware datetime comparison around ambiguous times

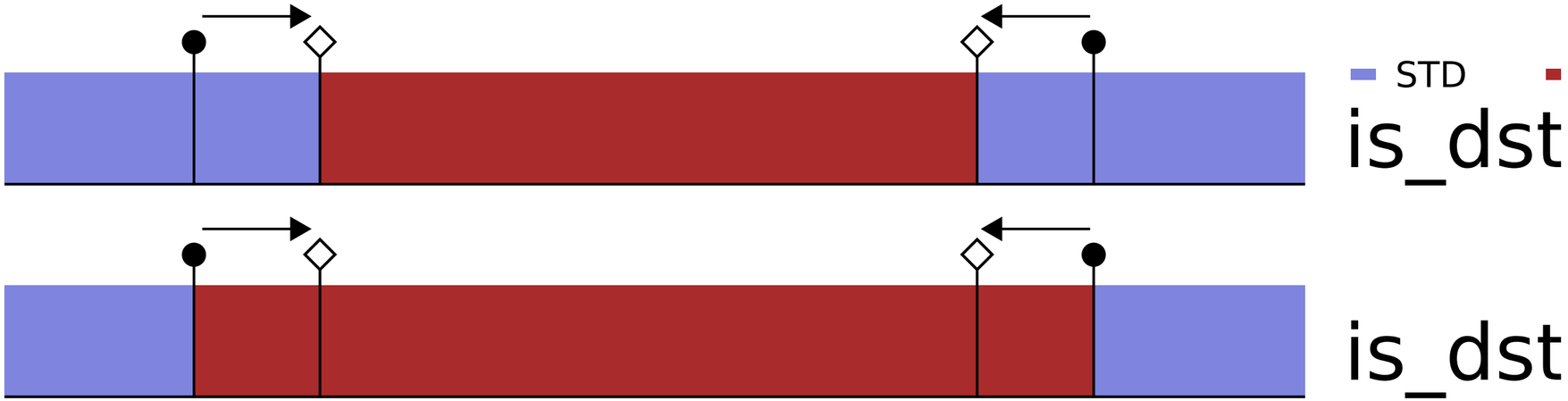
Whether you are on the fold side is a *property of the datetime*:

```
>>> print_tzinfo(datetime(2004, 10, 31, 1, 30, tzinfo=NYC))          # fold=0
2004-10-31 01:30:00-0400
    tzname:   EDT;      UTC Offset:  -4.00h;        DST:      1.0h

>>> print_tzinfo(datetime(2004, 10, 31, 1, 30, fold=1, tzinfo=NYC))
2004-10-31 01:30:00-0500
    tzname:   EST;      UTC Offset:  -5.00h;        DST:      0.0h
```

N.B.: `fold=1` represents the *second* instance of an ambiguous datetime.

@pganssle
ganssle.io

pytz



PEP 495

@pganssle
ganssle.io

Concrete Time Zones: Local time

- UTC / Fixed Offsets Added in 3.2
- Local time ⭘ Basically supported in 3.6+
- IANA Time Zones

Naïve datetimes are now considered system local times, and you can attach a fixed offset zone to the information:

```
>>> print(datetime(2023, 11, 4, 12).astimezone())
2023-11-04 12:00:00-04:00
>>> print(datetime(2023, 11, 5, 12).astimezone())
2023-11-05 12:00:00-05:00
```

Setting `fold` on a naïve datetime works:

```
>>> print(datetime(2023, 11, 5, 1, fold=0).astimezone())
2023-11-05 01:00:00-04:00
>>> print(datetime(2023, 11, 5, 1, fold=1).astimezone())
2023-11-05 01:00:00-05:00
```

See my blog posts: Why naïve times are local times and Stop using utcnow and utcfromt

▪ DST
= 0
= 1

@pganssle
ganssle.io

Concrete Time Zones: Local time

- UTC / Fixed Offsets Added in 3.2
- Local time ⭘ Basically supported in 3.6+
- IANA Time Zones ✘ (as of Python 3.8)

Naïve datetimes are now considered system local times, and you can attach a fixed offset zone to the information:

```
>>> print(datetime(2023, 11, 4, 12).astimezone())
2023-11-04 12:00:00-04:00
>>> print(datetime(2023, 11, 5, 12).astimezone())
2023-11-05 12:00:00-05:00
```

Setting `fold` on a naïve datetime works:

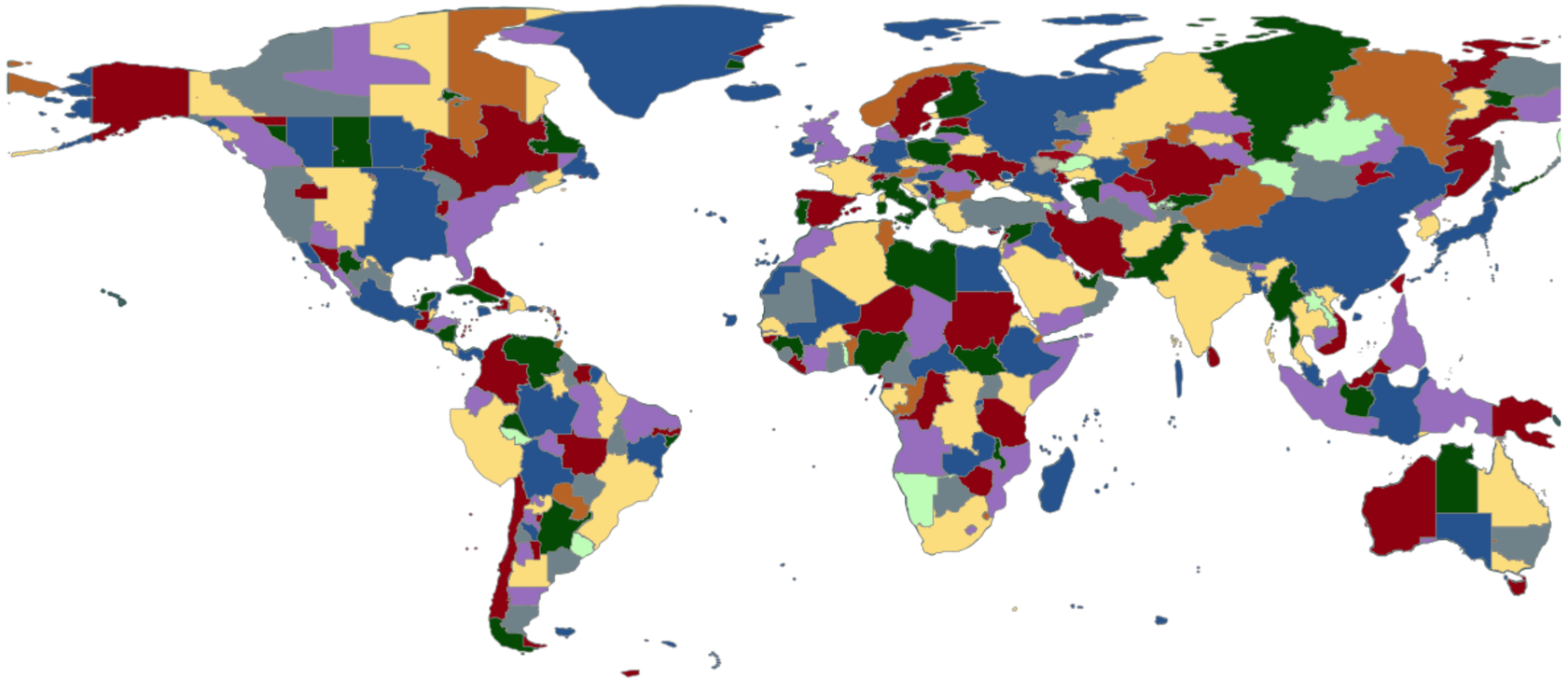
```
>>> print(datetime(2023, 11, 5, 1, fold=0).astimezone())
2023-11-05 01:00:00-04:00
>>> print(datetime(2023, 11, 5, 1, fold=1).astimezone())
2023-11-05 01:00:00-05:00
```

See my blog posts: Why naïve times are local times and Stop using utcnow and utcfromt

@pganssle
ganssle.io

data source: IANA Time Zones

- Provides historical time zone information
- Standard open source (public domain) source for time zone information
- Shipped with many operating systems
- Source for `dateutil` and `pytz`'s data.
- 2-21 releases per year (average 9)



@pganssle
ganssle.io

PEP 615: Support for the IANA Time Zone Database in the Sta

`zoneinfo` — IANA time zone support

New in version 3.9.

The `zoneinfo` module provides a concrete time zone implementation to support the IANA time zone database as originally specified in **PEP 615**. By default, `zoneinfo` uses the system's time zone data if available; if no system time zone data is available, the library will fall back to using the first-party `tzdata` package available on PyPI.

See also:

Module: `datetime`
Provides the `time` and `datetime` types with which the `ZoneInfo` class is designed to be used.

Package `tzdata`
First-party package maintained by the CPython core developers to supply time zone data via PyPI.

- When the system has IANA time zone data available, it is used
  - Defaults to well-known deployment locations
  - Configurable in-program using `zoneinfo.reset_tzpath`
  - Configurable with environment variable `PYTHONTZPATH`
  - Default can be set at compile time
- We also provide the `tzdata` package on PyPI — a "first party" data-only fallback library.

@pganssle
ganssle.io

A curious case...

```
>>> LON = ZoneInfo("Europe/London")

>>> x = datetime(2007, 3, 25, 1, 0, tzinfo=LON)
>>> ts = x.timestamp()
>>> y = datetime.fromtimestamp(ts, LON)
>>> z = datetime.fromtimestamp(ts, ZoneInfo.no_cache("Europe/London"))
```

@pganssle
ganssle.io

A curious case...

```
>>> LON = ZoneInfo("Europe/London")

>>> x = datetime(2007, 3, 25, 1, 0, tzinfo=LON)
>>> ts = x.timestamp()
>>> y = datetime.fromtimestamp(ts, LON)
>>> z = datetime.fromtimestamp(ts, ZoneInfo.no_cache("Europe/London"))
```

```
>>> x == y
False
```

@pganssle
ganssle.io

A curious case...

```
>>> LON = ZoneInfo("Europe/London")

>>> x = datetime(2007, 3, 25, 1, 0, tzinfo=LON)
>>> ts = x.timestamp()
>>> y = datetime.fromtimestamp(ts, LON)
>>> z = datetime.fromtimestamp(ts, ZoneInfo.no_cache("Europe/London"))
```

```
>>> x == y
False
```

```
>>> x == z
True
```

@pganssle
ganssle.io

A curious case...

```
>>> LON = ZoneInfo("Europe/London")

>>> x = datetime(2007, 3, 25, 1, 0, tzinfo=LON)
>>> ts = x.timestamp()
>>> y = datetime.fromtimestamp(ts, LON)
>>> z = datetime.fromtimestamp(ts, ZoneInfo.no_cache("Europe/London"))
```

```
>>> x == y
False
```

```
>>> x == z
True
```

```
>>> y == z
True
```

@pganssle
ganssle.io

Hint

`2007-03-25 01:00:00` is imaginary in London!

```
>>> print(x)                                # x (LON)
2007-03-25 01:00:00+01:00

>>> print(x.astimezone(timezone.utc))       # x (LON → UTC)
2007-03-25 00:00:00+00:00

>>> print(x.astimezone(timezone.utc).
...         .astimezone(LON))               # x (LON → UTC → LON)
2007-03-25 00:00:00+00:00
```

@pganssle
ganssle.io

What does equality mean?

1. Wall time semantics: compare only naïve portions

```
x == y  False
x == z  False
y == z  True
```

2. Absolute time semantics: convert to UTC

```
x == y  True
x == z  True
y == z  True
```

```
                Wa
x: 2007-03-2
y: 2007-03-2
z: 2007-03-2
```

@pganssle
ganssle.io

What does equality mean?

1. Wall time semantics: compare only naïve portions

`x == y`	`False`
`x == z`	`False`
`y == z`	`True`

2. Absolute time semantics: convert to UTC

`x == y`	`True`
`x == z`	`True`
`y == z`	`True`

```
x: 2007-03-2

y: 2007-03-2

z: 2007-03-2
```

@pganssle
ganssle.io

What does equality mean?

1. Wall time semantics: compare only naïve portions

`x == y`	`False`
`x == z`	`False`
`y == z`	`True`

```
x: 2007-03-2

y: 2007-03-2

z: 2007-03-2
```

2. Absolute time semantics: convert to UTC

`x == y`	`True`
`x == z`	`True`
`y == z`	`True`

Another hint

```
>>> x.tzinfo is y.tzinfo
True
>>> x.tzinfo is z.tzinfo
False
```

@pganssle
ganssle.io

Semantics of aware datetime comparison:

1. When two `datetimes` are in the *same zone*, only the naïve portion is compared (wall time sema
2. When they are in *different zones*, both are converted to UTC first, then compared (absolute time
3. Two `datetimes` are in the "same zone" only if `dt1.tzinfo is dt2.tzinfo`.

@pganssle
ganssle.io

Semantics of aware datetime comparison:

1. When two `datetimes` are in the *same zone*, only the naïve portion is compared (wall time sema
2. When they are in *different zones*, both are converted to UTC first, then compared (absolute time
3. Two `datetimes` are in the "same zone" only if `dt1.tzinfo is dt2.tzinfo`.

Mystery solved:

	Wall	Absolute	`datetime`
x == y	**False**	True	False
x == z	False	**True**	True
y == z	**True**	True	True

@pganssle
ganssle.io

`zoneinfo`: Cache behavior

Calls to the default constructor with identical arguments are guaranteed to return objects which c
specifically, the following must always be valid:

```
a = ZoneInfo(key)
b = ZoneInfo(key)
assert a is b
```

This is because `datetime` assumes that time zones are singletons, which would cause confusing resu
implementation:

```
>>> from datetime import *
>>> from simple_zoneinfo import SimpleZoneInfo
>>> dt0 = datetime(2020, 3, 8, tzinfo=SimpleZoneInfo("America/New_York"))
>>> dt1 = dt0 + timedelta(1)
>>> dt2 = dt1.replace(tzinfo=SimpleZoneInfo("America/New_York"))
>>> dt2 == dt1
True
>>> print(dt2 - dt1)
0:00:00
>>> print(dt2 - dt0)
23:00:00
>>> print(dt1 - dt0)
1 day, 0:00:00
```

See PEP 615 and the documentation for more information than you would ever want about work

Semantics of aware datetime arithmetic

An analogous problem for comparison semantics is that addition across a DST boundary is no

```
>>> NYC = ZoneInfo("America/New_York")
>>> dt1 = datetime(2020, 3, 7, 13, tzinfo=NYC)
>>> dt2 = d1 + timedelta(days=1)
```

Given that there is a DST transition between `dt1` and `dt2`, there are two option

```
>>> print(wall_add(dt1, timedelta(days=1)))  # Next calendar day at the same time
2020-03-08 13:00-04:00

>>> print(absolute_add(dt1, timedelta(days=1)))  # 24 elapsed hours after dt1
2020-03-08 12:00-04:00
```

@pganssle
ganssle.io

Semantics of aware datetime arithmetic

Datetime always uses wall-time semantics when interacting with a `timedelta`

```
>>> print(wall_add(dt1, timedelta(days=1)))
2020-03-08 13:00-04:00

>>> print(absolute_add(dt1, timedelta(days=1)))
2020-03-08 12:00-04:00

>>> print(dt1 + timedelta(days=1))
2020-03-08 13:00-04:00
```

When two `datetimes` are subtracted, the behavior is different for same-zone and different-zo

```
>>> dt2 = datetime(2020, 3, 8, 13, tzinfo=NYC)
>>> dt1_same = datetime(2020, 3, 7, 13, tzinfo=NYC)
>>> dt1_different = dt1_same.astimezone(timezone.utc)  # dt1_same == dt1_different!

>>> print(dt2 - dt1_same)
1 day, 0:00:00

>>> print(dt2 - dt1_different)
23:00:00
```

See my blog post "Semantics of timezone-aware datetime arithmetic" (https://blog.ganssle.io/artic
datetime-arithmetic.html) for a more thorough analysis.

@pganssle
ganssle.io

Using `zoneinfo`: Absolute time semantics

Many `pytz` users will be surprised by the "wall time" semantics of `datetime`. To deliberately use ab
convert to UTC first:

```
def absolute_add(dt: datetime, td: timedelta) -> datetime:
    dt_utc = dt.astimezone(timezone.utc)
    rv_utc = dt_utc + td
    return rv_utc.astimezone(dt.tzinfo)

def absolute_diff(dt1: datetime, dt2: datetime) -> timedelta:
    dt1_utc = dt1.astimezone(timezone.utc)
    dt2_utc = dt2.astimezone(timezone.utc)

    return dt1 - dt2
```

@pganssle
ganssle.io

Benefits of `zoneinfo`?

- Only major time zone library with year 2038 and slim tzdata support
- It's *fast* (numbers from `backports.zoneinfo`'s benchmark suite):

```
Running constructor in zone America/New_York
c_zoneinfo: mean: 214.65 ns ± 43.48 ns; min: 190.88 ns (k=5, N=1000000)
pytz: mean: 1.21 µs ± 78.31 ns; min: 1.10 µs (k=5, N=200000)
dateutil: mean: 1.33 µs ± 117.35 ns; min: 1.23 µs (k=5, N=200000)

Running from_utc in zone America/New_York
c_zoneinfo: mean: 658.55 ns ± 28.92 ns; min: 617.08 ns (k=5, N=500000)
pytz: mean: 5.12 µs ± 515.26 ns; min: 4.70 µs (k=5, N=50000)
dateutil: mean: 10.64 µs ± 746.99 ns; min: 10.20 µs (k=5, N=20000)

Running to_utc in zone America/New_York
c_zoneinfo: mean: 616.13 ns ± 16.14 ns; min: 604.76 ns (k=5, N=500000)
pytz: mean: 848.44 ns ± 28.10 ns; min: 806.72 ns (k=5, N=500000)
dateutil: mean: 8.03 µs ± 509.75 ns; min: 7.55 µs (k=5, N=50000)

Running utcoffset in zone America/New_York
c_zoneinfo: mean: 373.89 ns ± 5.76 ns; min: 368.24 ns (k=5, N=1000000)
pytz: mean: 564.55 ns ± 13.65 ns; min: 552.88 ns (k=5, N=500000)
dateutil: mean: 7.95 µs ± 642.62 ns; min: 7.44 µs (k=5, N=50000)
```

Because of the C backend, `zoneinfo` is faster than `pytz` and `dateutil` on every n

@pganssle
ganssle.io

Migrating from `pytz`

If you have any public-facing interface that returns `pytz` timezones (or `datetimes` localized with py

change to move away from `pytz`:

```
def pre_migration():
    return pytz.timezone("America/New_York").localize(datetime(2020, 1, 1))

def post_migration():
    return datetime(2020, 1, 1, tzinfo=ZoneInfo("America/New_York"))

def sixty_days_later(dt: datetime) -> datetime:
    non_normalized_dt = dt + timedelta(days=60)
    return dt.tzinfo.normalize(non_normalized_dt)
```

```
>>> sixty_days_later(pre_migration())
datetime.datetime(2020, 3, 1, 0, 0, tzinfo=<DstTzInfo 'America/New_York' EST-1 day, 19:00:00 STD>)

>>> sixty_days_later(post_migration())
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-7-b71365e0022f> in <module>
----> 1 sixty_days_later(post_migration())

<ipython-input-5-f165abf34e2a> in sixty_days_later(dt)
      7 def sixty_days_later(dt: datetime) -> datetime:
      8     non_normalized_dt = dt + timedelta(days=60)
----> 9     return dt.tzinfo.normalize(non_normalized_dt)

AttributeError: 'zoneinfo.ZoneInfo' object has no attribute 'normalize'
```

@pganssle
ganssle.io

pytz-deprecation-shim

pytz-deprecation-shim is a mostly backwards-compatible implementation of pytz's interfac
wrapper around zoneinfo. It can be used exactly as a zoneinfo.ZoneInfo object w

```
>>> import pytz_deprecation_shim as pds
>>> from datetime import datetime, timedelta
>>> LA = pds.timezone("America/Los_Angeles")

>>> dt = datetime(2020, 10, 31, 12, tzinfo=LA)
>>> print(dt)
2020-10-31 12:00:00-07:00

>>> dt.tzname()
'PDT'
```

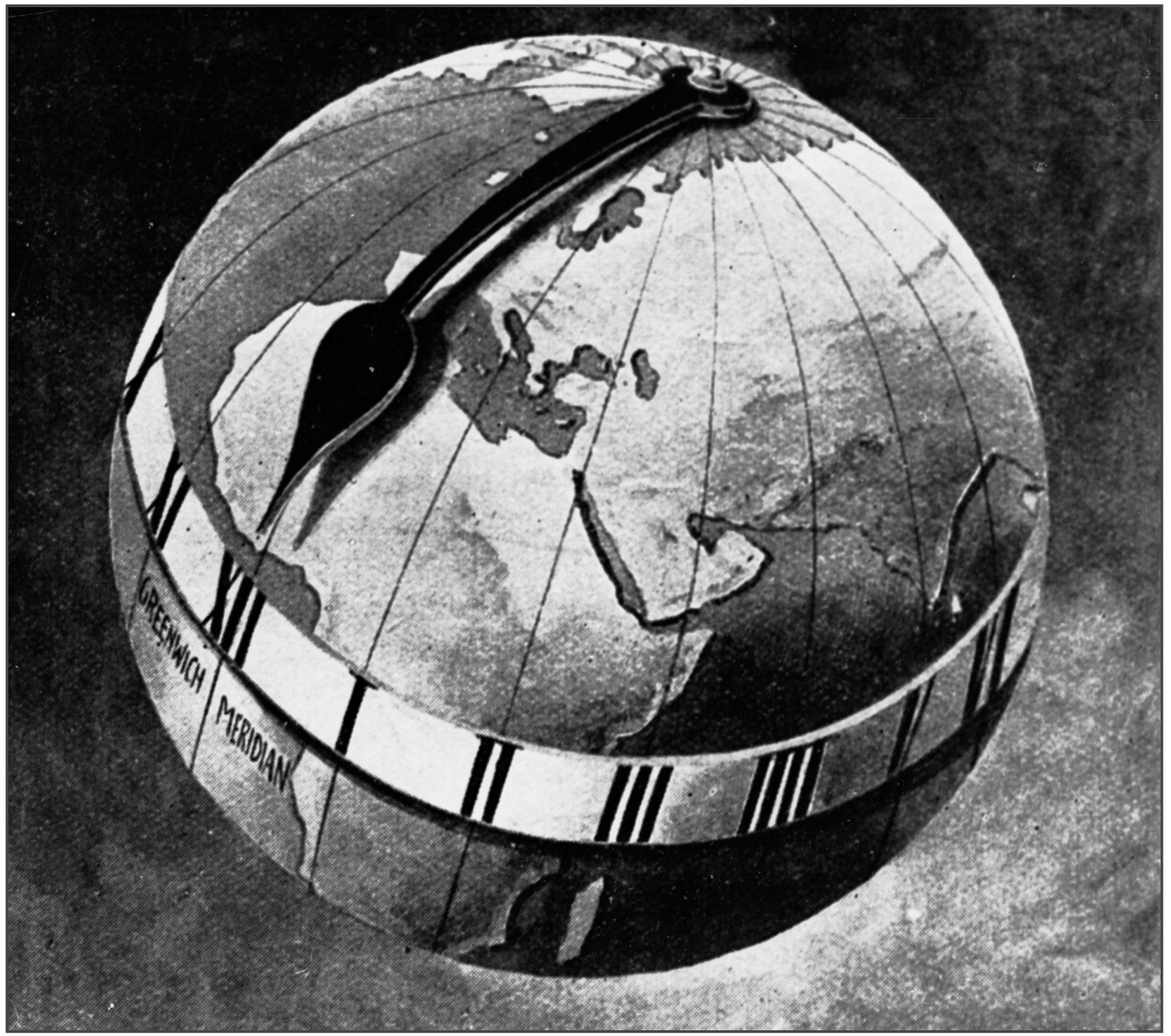
But also exposes pytz's interface, raising a DeprecationWarning when pytz-specific fea

```
>>> dt = LA.localize(datetime(2020, 10, 31, 12))
<stdin>:1: PytzUsageWarning: The localize method is no longer necessary, as
this time zone supports the fold attribute (PEP 495). For more details on
migrating to a PEP 495-compliant implementation, see
https://pytz-deprecation-shim.readthedocs.io/en/latest/migration.html

 >>> print(dt)
2020-10-31 12:00:00-07:00
>>> dt.tzname()
'PDT'
```

Caution: There are some changes in arithmetic semantics, see the migration guid

@pganssle
ganssle.io

Thank you



@pganssle
ganssle.io