Python Datetime TZ Documentation

Release 0.2

Tim 'mithro' Ansell

January 27, 2016

Contents

A version of the datetime module which *cares* about timezones.

This module will never return a naive datetime object. This requires the module know your local timezone, which it tries really hard to figure out.

You can override the detection by using the datetime.tzaware.defaulttz_set method. It the module is unable to figure out the timezone itself this method *must* be called before the normal module is imported. If done before importing it can also speed up the time taken to import as the defaulttz will no longer try and do the detection.

class `datetime_tz.`**`datetime_tz`**

An extension of the inbuilt datetime adding more functionality.

The extra functionality includes:

- Partial parsing support (IE 2006/02/30 matches %Y/%M/%D %H:%M)
- Full integration with pytz (just give it the string of the timezone!)
- Proper support for going to/from Unix timestamps (which are in UTC!).

`asdate`()

Return this datetime_tz as a date object.

Returns: This datetime_tz as a date object.

`asdatetime`(*naive=True*)

Return this datetime_tz as a datetime object.

Args: naive: Return *without* any tz info.

Returns: This datetime_tz as a datetime object.

`astimezone`(*tzinfo*)

Returns a version of this timestamp converted to the given timezone.

Args:

tzinfo: Either a datetime.tzinfo object or a string (which will be looked up in pytz.

Returns: A datetime_tz object in the given timezone.

classmethod `combine`(*date*, *time*, *tzinfo=None*)

date, time, [tz] -> datetime with same date and time fields.

classmethod `fromtimestamp`(*timestamp*)

Returns a datetime object of a given timestamp (in local tz).

classmethod `now`(*tzinfo=None*)

[tz] -> new datetime with tz's local day and time.

`replace`(***kw*)

Return datetime with new specified fields given as arguments.

For example, dt.replace(days=4) would return a new datetime_tz object with exactly the same as dt but with the days attribute equal to 4.

Any attribute can be replaced, but tzinfo can not be set to None.

Args: Any datetime_tz attribute.

Returns: A datetime_tz object with the attributes replaced.

Raises: TypeError: If the given replacement is invalid.

classmethod `smartparse`(*toparse*, *tzinfo=None*)

Method which uses dateutil.parse and extras to try and parse the string.

Valid dates are found at: http://labix.org/python-dateutil#head-1443e0f14ad5dff07efd465e080d1110920673d8-2

Other valid formats include: "now" or "today" "yesterday" "tomorrow" "5 minutes ago" "10 hours ago" "10h5m ago" "start of yesterday" "end of tomorrow" "end of 3rd of March"

Args: toparse: The string to parse. tzinfo: Timezone for the resultant datetime_tz object should be in.

(Defaults to your local timezone.)

Returns: New datetime_tz object.

Raises: ValueError: If unable to make sense of the input.

classmethod **today**(*tzinfo=None*)
[tz] -> new datetime with tz's local day and time.

totimestamp()
Convert this datetime object back to a unix timestamp.

The Unix epoch is the time 00:00:00 UTC on January 1, 1970.

Returns: Unix timestamp.

classmethod **utcfromtimestamp**(*timestamp*)
Returns a datetime object of a given timestamp (in UTC).

classmethod **utcnow**()
Return a new datetime representing UTC day and time.

datetime_tz.**detect_timezone**()
Try and detect the timezone that Python is currently running in.

We have a bunch of different methods for trying to figure this out (listed in order they are attempted).

- In windows, use win32timezone.TimeZoneInfo.local()
- Try TZ environment variable.
- Try and find /etc/timezone file (with timezone name).
- Try and find /etc/localtime file (with timezone data).
- Try and match a TZ to the current dst/offset/shortname.

Returns: The detected local timezone as a tzinfo object

Raises: pytz.UnknownTimeZoneError: If it was unable to detect a timezone.

class datetime_tz.**iterate**
Helpful iterators for working with datetime_tz objects.

static **between**(*start*, *delta*, *end=None*)
Return an iterator between this date till given end point.

Example usage:

```
>>> d = datetime_tz.smartparse("5 days ago")
2008/05/12 11:45
>>> for i in d.between(timedelta(days=1), datetime_tz.now()):
>>>    print i
2008/05/12 11:45
2008/05/13 11:45
2008/05/14 11:45
```

```
            2008/05/15 11:45
            2008/05/16 11:45
```

Args: start: The date to start at. delta: The interval to iterate with. end: (Optional) Date to end at. If not given the iterator will never

terminate.

Yields: datetime_tz objects.

static `days`(*start*, *end=None*)
Iterate over the days between the given datetime_tzs.

Args: start: datetime_tz to start from. end: (Optional) Date to end at, if not given the iterator will never

terminate.

Returns: An iterator which generates datetime_tz objects a day apart.

static `hours`(*start*, *end=None*)
Iterate over the hours between the given datetime_tzs.

Args: start: datetime_tz to start from. end: (Optional) Date to end at, if not given the iterator will never

terminate.

Returns: An iterator which generates datetime_tz objects a hour apart.

static `minutes`(*start*, *end=None*)
Iterate over the minutes between the given datetime_tzs.

Args: start: datetime_tz to start from. end: (Optional) Date to end at, if not given the iterator will never

terminate.

Returns: An iterator which generates datetime_tz objects a minute apart.

static `seconds`(*start*, *end=None*)
Iterate over the seconds between the given datetime_tzs.

Args: start: datetime_tz to start from. end: (Optional) Date to end at, if not given the iterator will never

terminate.

Returns: An iterator which generates datetime_tz objects a second apart.

static `weeks`(*start*, *end=None*)
Iterate over the weeks between the given datetime_tzs.

Args: start: datetime_tz to start from. end: (Optional) Date to end at, if not given the iterator will never

terminate.

Returns: An iterator which generates datetime_tz objects a week apart.

`datetime_tz.`**`localtz`**()
Get the local timezone.

Returns: The localtime timezone as a tzinfo object.

`datetime_tz.`**`localtz_set`**(*timezone*)
Set the local timezone.

class `datetime_tz.`**`timedelta`**
Difference between two datetime values.

days
Number of days.

microseconds
Number of microseconds (>= 0 and less than 1 second).

seconds
Number of seconds (>= 0 and less than 1 day).

total_seconds()
Total seconds in the duration.

datetime_tz.**localize**(*dt*, *force_to_local=True*)
Localize a datetime to the local timezone.

If dt is naive, returns the same datetime with the local timezone, otherwise uses astimezone to convert.

Args: dt: datetime object. force_to_local: Force all results to be in local time.

Returns: A datetime_tz object.

datetime_tz.**get_naive**(*dt*)
Gets a naive datetime from a datetime.

datetime_tz objects can't just have tzinfo replaced with None, you need to call asdatetime.

Args: dt: datetime object.

Returns: datetime object without any timezone information.

datetime_tz.**localtz_name**()
Returns the name of the local timezone.

datetime_tz.**require_timezone**(*zone*)
Raises an AssertionError if we are not in the correct timezone.

CHAPTER 1

pytz_abbr

Common time zone acronyms/abbreviations for use with the datetime_tz module.

WARNING: There are lots of caveats when using this module which are listed below.

CAVEAT 1: The acronyms/abbreviations are not globally unique, they are not even unique within a region. For example, EST can mean any of,

> Eastern Standard Time in Australia (which is 10 hour ahead of UTC) Eastern Standard Time in North America (which is 5 hours behind UTC)

Where there are two abbreviations the more popular one will appear in the all dictionary, while the less common one will only appear in that countries region dictionary. IE If using all, EST will be mapped to Eastern Standard Time in North America.

CAVEAT 2: Many of the acronyms don't map to a neat Oslon timezones. For example, Eastern European Summer Time (EEDT) is used by many different countries in Europe *at different times*! If the acronym does not map neatly to one zone it is mapped to the Etc/GMT+-XX Oslon zone. This means that any date manipulations can end up with idiot things like summer time in the middle of winter.

CAVEAT 3: The Summer/Standard time difference is really important! For an hour each year it is needed to determine which time you are actually talking about.

> 2002-10-27 01:20:00 EST != 2002-10-27 01:20:00 EDT

class datetime_tz.pytz_abbr.**tzabbr**
A timezone abbreviation.

WARNING: This is not a tzinfo implementation! Trying to use this as tzinfo object will result in failure. We inherit from datetime.tzinfo so we can get through the dateutil checks.

datetime_tz.pytz_abbr.**tzabbr_register**(*abbr*, *name*, *region*, *zone*, *dst*)
Register a new timezone abbreviation in the global registry.

If another abbreviation with the same name has already been registered it new abbreviation will only be registered in region specific dictionary.

d