dateutil Documentation

Release 2.6.1

dateutil

Jul 10, 2017

Contents

The *dateutil* module provides powerful extensions to the standard *datetime* module, available in Python.

CHAPTER 1

Download

dateutil is available on PyPI https://pypi.python.org/pypi/python-dateutil/

The documentation is hosted at: https://dateutil.readthedocs.io/

CHAPTER 2

Code

https://github.com/dateutil/dateutil/

CHAPTER 3

Features

- Computing of relative deltas (next month, next year, next monday, last week of month, etc);
- Computing of relative deltas between two given date and/or datetime objects;
- Computing of dates based on very flexible recurrence rules, using a superset of the iCalendar specification. Parsing of RFC strings is supported as well.
- Generic parsing of dates in almost any string format;
- Timezone (tzinfo) implementations for tzfile(5) format files (/etc/localtime, /usr/share/zoneinfo, etc), TZ environment string (in all known formats), iCalendar format files, given ranges (with help from relative deltas), local machine timezone, fixed offset timezone, UTC timezone, and Windows registry-based time zones.
- Internal up-to-date world timezone information based on Olson's database.
- Computing of Easter Sunday dates for any given year, using Western, Orthodox or Julian algorithms;
- A comprehensive test suite.

CHAPTER 4

Quick example

Here's a snapshot, just to give an idea about the power of the package. For more examples, look at the documentation.

Suppose you want to know how much time is left, in years/months/days/etc, before the next easter happening on a year with a Friday 13th in August, and you want to get today's date out of the "date" unix system command. Here is the code:

```
>>> from dateutil.relativedelta import *
>>> from dateutil.easter import *
>>> from dateutil.rrule import *
>>> from dateutil.parser import *
>>> from datetime import *
>>> now = parse("Sat Oct 11 17:13:46 UTC 2003")
>>> today = now.date()
>>> year = rrule(YEARLY,dtstart=now,bymonth=8,bymonthday=13,byweekday=FR)[0].year
>>> rdelta = relativedelta(easter(year), today)
>>> print("Today is: %s" % today)
Today is: 2003-10-11
>>> print("Year with next Aug 13th on a Friday is: %s" % year)
Year with next Aug 13th on a Friday is: 2004
>>> print("How far is the Easter of that year: %s" % rdelta)
How far is the Easter of that year: relativedelta(months=+6)
>>> print("And the Easter of that year is: %s" % (today+rdelta))
And the Easter of that year is: 2004-04-11
```

Being exactly 6 months ahead was **really** a coincidence :)

CHAPTER 5

Author

The dateutil module was written by Gustavo Niemeyer <gustavo@niemeyer.net> in 2003.

It is maintained by:

- Gustavo Niemeyer <gustavo@niemeyer.net> 2003-2011
- Tomi Pieviläinen <tomi.pievilainen@iki.fi> 2012-2014
- Yaron de Leeuw <me@jarondl.net> 2014-2016
- Paul Ganssle <paul@ganssle.io> 2015-

Our mailing list is available at dateutil@python.org. As it is hosted by the PSF, it is subject to the PSF code of conduct.

CHAPTER 6

Building and releasing

When you get the source, it does not contain the internal zoneinfo database. To get (and update) the database, run the updatezinfo.py script. Make sure that the zic command is in your path, and that you have network connectivity to get the latest timezone information from IANA, or from our mirror of the IANA database.

Starting with version 2.4.1, all source and binary distributions will be signed by a PGP key that has, at the very least, been signed by the key which made the previous release. A table of release signing keys can be found below:

Releases	Signing key fingerprint
2.4.1-	6B49 ACBA DCF6 BD1C A206 67AB CD54 FCE3 D964 BEFB

CHAPTER 7

Testing

dateutil has a comprehensive test suite, which can be run simply by running *python setup.py test [-q]* in the project root. Note that if you don't have the internal zoneinfo database, some tests will fail. Apart from that, all tests should pass.

To easily test dateutil against all supported Python versions, you can use tox.

All github pull requests are automatically tested using travis and appveyor.

CHAPTER 8

Documentation

Contents:

easter

This module offers a generic easter computing method for any given year, using Western, Orthodox or Julian algorithms.

`dateutil.easter.`**`easter`**(*year*, *method=3*)

This method was ported from the work done by GM Arts, on top of the algorithm by Claus Tondering, which was based in part on the algorithm of Ouding (1940), as quoted in "Explanatory Supplement to the Astronomical Almanac", P. Kenneth Seidelmann, editor.

This algorithm implements three different easter calculation methods:

1 - Original calculation in Julian calendar, valid in dates after 326 AD

2 - Original method, with date converted to Gregorian calendar, valid in years 1583 to 4099

3 - Revised method, in Gregorian calendar, valid in years 1583 to 4099 as well

These methods are represented by the constants:

- `EASTER_JULIAN = 1`
- `EASTER_ORTHODOX = 2`
- `EASTER_WESTERN = 3`

The default method is method 3.

More about the algorithm may be found at:

http://users.chariot.net.au/~gmarts/eastalg.htm

and

http://www.tondering.dk/claus/calendar.html

parser

This module offers a generic date/time string parser which is able to parse most known formats to represent a date and/or time.

This module attempts to be forgiving with regards to unlikely input formats, returning a datetime object even for dates which are ambiguous. If an element of a date/time stamp is omitted, the following rules are applied: - If AM or PM is left unspecified, a 24-hour clock is assumed, however, an hour

on a 12-hour clock (`0 <= hour <= 12`) *must* be specified if AM or PM is specified.

- If a time zone is omitted, a timezone-naive datetime is returned.

If any other elements are missing, they are taken from the `datetime.datetime` object passed to the parameter `default`. If this results in a day number exceeding the valid number of days per month, the value falls back to the end of the month.

Additional resources about date/time string formats can be found below:

- A summary of the international standard date and time notation
- W3C Date and Time Formats
- Time Formats (Planetary Rings Node)
- CPAN ParseDate module
- Java SimpleDateFormat Class

`dateutil.parser.`**`parse`**(*timestr*, *parserinfo=None*, ***kwargs*)

Parse a string in one of the supported formats, using the `parserinfo` parameters.

Parameters

- **`timestr`** – A string containing a date/time stamp.
- **`parserinfo`** – A *`parserinfo`* object containing parameters for the parser. If `None`, the default arguments to the *`parserinfo`* constructor are used.

The `**kwargs` parameter takes the following keyword arguments:

Parameters

- **`default`** – The default datetime object, if this is a datetime object and not `None`, elements specified in `timestr` replace elements in the default object.
- **`ignoretz`** – If set `True`, time zones in parsed strings are ignored and a naive `datetime` object is returned.
- **`tzinfos`** – Additional time zone names / aliases which may be present in the string. This argument maps time zone names (and optionally offsets from those time zones) to time zones. This parameter can be a dictionary with timezone aliases mapping time zone names to time zones or a function taking two parameters (`tzname` and `tzoffset`) and returning a time zone.

  The timezones to which the names are mapped can be an integer offset from UTC in minutes or a `tzinfo` object.

```
>>> from dateutil.parser import parse
>>> from dateutil.tz import gettz
>>> tzinfos = {"BRST": -10800, "CST": gettz("America/Chicago")}
>>> parse("2012-01-19 17:21:00 BRST", tzinfos=tzinfos)
datetime.datetime(2012, 1, 19, 17, 21, tzinfo=tzoffset(u'BRST', -
↪10800))
```

```
>>> parse("2012-01-19 17:21:00 CST", tzinfos=tzinfos)
datetime.datetime(2012, 1, 19, 17, 21,
                  tzinfo=tzfile('/usr/share/zoneinfo/America/
↪Chicago'))
```

This parameter is ignored if `ignoretz` is set.

- **dayfirst** – Whether to interpret the first value in an ambiguous 3-integer date (e.g. 01/05/09) as the day (`True`) or month (`False`). If `yearfirst` is set to `True`, this distinguishes between YDM and YMD. If set to `None`, this value is retrieved from the current *parserinfo* object (which itself defaults to `False`).
- **yearfirst** – Whether to interpret the first value in an ambiguous 3-integer date (e.g. 01/05/09) as the year. If `True`, the first number is taken to be the year, otherwise the last number is taken to be the year. If this is set to `None`, the value is retrieved from the current *parserinfo* object (which itself defaults to `False`).
- **fuzzy** – Whether to allow fuzzy parsing, allowing for string like "Today is January 1, 2047 at 8:21:00AM".
- **fuzzy_with_tokens** – If `True`, `fuzzy` is automatically set to True, and the parser will return a tuple where the first element is the parsed `datetime.datetime` datetimestamp and the second element is a tuple containing the portions of the string which were ignored:

```
>>> from dateutil.parser import parse
>>> parse("Today is January 1, 2047 at 8:21:00AM", fuzzy_with_
↪tokens=True)
(datetime.datetime(2047, 1, 1, 8, 21), (u'Today is ', u' ', u'at
↪'))
```

Returns Returns a `datetime.datetime` object or, if the `fuzzy_with_tokens` option is `True`, returns a tuple, the first element being a `datetime.datetime` object, the second a tuple containing the fuzzy tokens.

Raises

- **ValueError** – Raised for invalid or unknown string format, if the provided `tzinfo` is not in a valid format, or if an invalid date would be created.
- **OverflowError** – Raised if the parsed date exceeds the largest valid C integer on your system.

class `dateutil.parser.`**parserinfo**(*dayfirst=False*, *yearfirst=False*)

Class which handles what inputs are accepted. Subclass this to customize the language and acceptable values for each parameter.

Parameters

- **dayfirst** – Whether to interpret the first value in an ambiguous 3-integer date (e.g. 01/05/09) as the day (`True`) or month (`False`). If `yearfirst` is set to `True`, this distinguishes between YDM and YMD. Default is `False`.
- **yearfirst** – Whether to interpret the first value in an ambiguous 3-integer date (e.g. 01/05/09) as the year. If `True`, the first number is taken to be the year, otherwise the last number is taken to be the year. Default is `False`.

AMPM = [('am', 'a'), ('pm', 'p')]

HMS = [('h', 'hour', 'hours'), ('m', 'minute', 'minutes'), ('s', 'second', 'seconds')]

JUMP = [' ', '.', ',', ';', '-', '/', '"', 'at', 'on', 'and', 'ad', 'm', 't', 'of', 'st', 'nd', 'rd', 'th']

MONTHS = [('Jan', 'January'), ('Feb', 'February'), ('Mar', 'March'), ('Apr', 'April'), ('May', 'May'), ('Jun', 'June'), ('Jul

PERTAIN = ['of']

TZOFFSET = {}

UTCZONE = ['UTC', 'GMT', 'Z']

WEEKDAYS = [('Mon', 'Monday'), ('Tue', 'Tuesday'), ('Wed', 'Wednesday'), ('Thu', 'Thursday'), ('Fri', 'Friday'), ('Sat',

ampm(*name*)

convertyear(*year*, *century_specified=False*)

hms(*name*)

jump(*name*)

month(*name*)

pertain(*name*)

tzoffset(*name*)

utczone(*name*)

validate(*res*)

weekday(*name*)

relativedelta

class dateutil.relativedelta.**relativedelta**(*dt1=None*, *dt2=None*, *years=0*, *months=0*, *days=0*, *leapdays=0*, *weeks=0*, *hours=0*, *minutes=0*, *seconds=0*, *microseconds=0*, *year=None*, *month=None*, *day=None*, *weekday=None*, *yearday=None*, *nlyearday=None*, *hour=None*, *minute=None*, *second=None*, *microsecond=None*)

The relativedelta type is based on the specification of the excellent work done by M.-A. Lemburg in his mx.DateTime extension. However, notice that this type does *NOT* implement the same algorithm as his work. Do *NOT* expect it to behave like mx.DateTime's counterpart.

There are two different ways to build a relativedelta instance. The first one is passing it two date/datetime classes:

```
relativedelta(datetime1, datetime2)
```

The second one is passing it any number of the following keyword arguments:

```
relativedelta(arg1=x,arg2=y,arg3=z...)

year, month, day, hour, minute, second, microsecond:
    Absolute information (argument is singular); adding or subtracting a
    relativedelta with absolute information does not perform an aritmetic
    operation, but rather REPLACES the corresponding value in the
    original datetime with the value(s) in relativedelta.

years, months, weeks, days, hours, minutes, seconds, microseconds:
    Relative information, may be negative (argument is plural); adding
    or subtracting a relativedelta with relative information performs
```

```
    the corresponding aritmetic operation on the original datetime value
    with the information in the relativedelta.

weekday:
    One of the weekday instances (MO, TU, etc). These instances may
    receive a parameter N, specifying the Nth weekday, which could
    be positive or negative (like MO(+1) or MO(-2). Not specifying
    it is the same as specifying +1. You can also use an integer,
    where 0=MO.

leapdays:
    Will add given days to the date found, if year is a leap
    year, and the date found is post 28 of february.

yearday, nlyearday:
    Set the yearday or the non-leap year day (jump leap days).
    These are converted to day/month/leapdays information.
```

Here is the behavior of operations with relativedelta:

1. Calculate the absolute year, using the 'year' argument, or the original datetime year, if the argument is not present.
2. Add the relative 'years' argument to the absolute year.
3. Do steps 1 and 2 for month/months.
4. Calculate the absolute day, using the 'day' argument, or the original datetime day, if the argument is not present. Then, subtract from the day until it fits in the year and month found after their operations.
5. Add the relative 'days' argument to the absolute day. Notice that the 'weeks' argument is multiplied by 7 and added to 'days'.
6. Do steps 1 and 2 for hour/hours, minute/minutes, second/seconds, microsecond/microseconds.
7. If the 'weekday' argument is present, calculate the weekday, with the given (wday, nth) tuple. wday is the index of the weekday (0-6, 0=Mon), and nth is the number of weeks to add forward or backward, depending on its signal. Notice that if the calculated date is already Monday, for example, using (0, 1) or (0, -1) won't change the day.

normalized()

Return a version of this object represented entirely using integer values for the relative attributes.

```
>>> relativedelta(days=1.5, hours=2).normalized()
relativedelta(days=1, hours=14)
```

Returns Returns a *dateutil.relativedelta.relativedelta* object.

weeks

rrule

The rrule module offers a small, complete, and very fast, implementation of the recurrence rules documented in the iCalendar RFC, including support for caching of results.

class dateutil.rrule.**rrule**(*freq*, *dtstart=None*, *interval=1*, *wkst=None*, *count=None*, *until=None*, *bysetpos=None*, *bymonth=None*, *bymonthday=None*, *byyearday=None*, *byeaster=None*, *byweekno=None*, *byweekday=None*, *byhour=None*, *byminute=None*, *bysecond=None*, *cache=False*)

That's the base of the rrule operation. It accepts all the keywords defined in the RFC as its constructor parameters (except byday, which was renamed to byweekday) and more. The constructor prototype is:

```
rrule(freq)
```

Where freq must be one of YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY, or SECONDLY.

Note: Per RFC section 3.3.10, recurrence instances falling on invalid dates and times are ignored rather than coerced:

> Recurrence rules may generate recurrence instances with an invalid date (e.g., February 30) or nonexistent local time (e.g., 1:30 AM on a day where the local time is moved forward by an hour at 1:00 AM). Such recurrence instances MUST be ignored and MUST NOT be counted as part of the recurrence set.

This can lead to possibly surprising behavior when, for example, the start date occurs at the end of the month:

```
>>> from dateutil.rrule import rrule, MONTHLY
>>> from datetime import datetime
>>> start_date = datetime(2014, 12, 31)
>>> list(rrule(freq=MONTHLY, count=4, dtstart=start_date))
...
[datetime.datetime(2014, 12, 31, 0, 0),
 datetime.datetime(2015, 1, 31, 0, 0),
 datetime.datetime(2015, 3, 31, 0, 0),
 datetime.datetime(2015, 5, 31, 0, 0)]
```

Additionally, it supports the following keyword arguments:

Parameters

- **cache** – If given, it must be a boolean value specifying to enable or disable caching of results. If you will use the same rrule instance multiple times, enabling caching will improve the performance considerably.
- **dtstart** – The recurrence start. Besides being the base for the recurrence, missing parameters in the final recurrence instances will also be extracted from this date. If not given, datetime.now() will be used instead.
- **interval** – The interval between each freq iteration. For example, when using YEARLY, an interval of 2 means once every two years, but with HOURLY, it means once every two hours. The default interval is 1.
- **wkst** – The week start day. Must be one of the MO, TU, WE constants, or an integer, specifying the first day of the week. This will affect recurrences based on weekly periods. The default week start is got from calendar.firstweekday(), and may be modified by calendar.setfirstweekday().
- **count** – How many occurrences will be generated.

  Note: As of version 2.5.0, the use of the `until` keyword together with the `count` keyword is deprecated per RFC-2445 Sec. 4.3.10.

- **until** – If given, this must be a datetime instance, that will specify the limit of the recurrence. The last recurrence in the rule is the greatest datetime that is less than or equal to the value specified in the `until` parameter.

  Note: As of version 2.5.0, the use of the `until` keyword together with the `count` keyword is deprecated per RFC-2445 Sec. 4.3.10.

- **bysetpos** – If given, it must be either an integer, or a sequence of integers, positive or negative. Each given integer will specify an occurrence number, corresponding to the nth occurrence of the rule inside the frequency period. For example, a bysetpos of -1 if combined with a MONTHLY frequency, and a byweekday of (MO, TU, WE, TH, FR), will result in the last work day of every month.
- **bymonth** – If given, it must be either an integer, or a sequence of integers, meaning the months to apply the recurrence to.
- **bymonthday** – If given, it must be either an integer, or a sequence of integers, meaning the month days to apply the recurrence to.
- **byyearday** – If given, it must be either an integer, or a sequence of integers, meaning the year days to apply the recurrence to.
- **byweekno** – If given, it must be either an integer, or a sequence of integers, meaning the week numbers to apply the recurrence to. Week numbers have the meaning described in ISO8601, that is, the first week of the year is that containing at least four days of the new year.
- **byweekday** – If given, it must be either an integer (0 == MO), a sequence of integers, one of the weekday constants (MO, TU, etc), or a sequence of these constants. When given, these variables will define the weekdays where the recurrence will be applied. It's also possible to use an argument n for the weekday instances, which will mean the nth occurrence of this weekday in the period. For example, with MONTHLY, or with YEARLY and BYMONTH, using FR(+1) in byweekday will specify the first friday of the month where the recurrence happens. Notice that in the RFC documentation, this is specified as BYDAY, but was renamed to avoid the ambiguity of that keyword.
- **byhour** – If given, it must be either an integer, or a sequence of integers, meaning the hours to apply the recurrence to.
- **byminute** – If given, it must be either an integer, or a sequence of integers, meaning the minutes to apply the recurrence to.
- **bysecond** – If given, it must be either an integer, or a sequence of integers, meaning the seconds to apply the recurrence to.
- **byeaster** – If given, it must be either an integer, or a sequence of integers, positive or negative. Each integer will define an offset from the Easter Sunday. Passing the offset 0 to byeaster will yield the Easter Sunday itself. This is an extension to the RFC specification.

replace(***kwargs*)
Return new rrule with same attributes except for those attributes given new values by whichever keyword arguments are specified.

class dateutil.rrule.**rruleset**(*cache=False*)
The rruleset type allows more complex recurrence setups, mixing multiple rules, dates, exclusion rules, and exclusion dates. The type constructor takes the following keyword arguments:

Parameters **cache** – If True, caching of results will be enabled, improving performance of multiple queries considerably.

exdate(**args*, ***kwargs*)

exrule(**args*, ***kwargs*)

rdate(**args*, ***kwargs*)

rrule(**args*, ***kwargs*)

tz

class dateutil.tz.**tzutc**
This is a tzinfo object that represents the UTC time zone.

dst(*dt*)

fromutc(*dt*)
Fast track version of fromutc() returns the original dt object for any valid datetime.datetime object.

is_ambiguous(*dt*)
Whether or not the "wall time" of a given datetime is ambiguous in this zone.

Parameters **dt** – A datetime.datetime, naive or time zone aware.

Returns Returns True if ambiguous, False otherwise.

New in version 2.6.0.

tzname(**args*, ***kwargs*)

utcoffset(*dt*)

class dateutil.tz.**tzoffset**(*name*, *offset*)
A simple class for representing a fixed offset from UTC.

Parameters

- **name** – The timezone name, to be returned when tzname() is called.
- **offset** – The time zone offset in seconds, or (since version 2.6.0, represented as a datetime.timedelta object.

dst(*dt*)

fromutc(*dt*)

is_ambiguous(*dt*)
Whether or not the "wall time" of a given datetime is ambiguous in this zone.

Parameters **dt** – A datetime.datetime, naive or time zone aware.

Returns Returns True if ambiguous, False otherwise.

New in version 2.6.0.

tzname(**args*, ***kwargs*)

utcoffset(*dt*)

class dateutil.tz.**tzlocal**
A tzinfo subclass built around the time timezone functions.

dst(*dt*)

is_ambiguous(*dt*)
Whether or not the "wall time" of a given datetime is ambiguous in this zone.

Parameters **dt** – A `datetime.datetime`, naive or time zone aware.

Returns Returns `True` if ambiguous, `False` otherwise.

New in version 2.6.0.

tzname(**args*, ***kwargs*)

utcoffset(*dt*)

class `dateutil.tz.`**tzfile**(*fileobj*, *filename=None*)

This is a `tzinfo` subclass thant allows one to use the `tzfile(5)` format timezone files to extract current and historical zone information.

Parameters

- **fileobj** – This can be an opened file stream or a file name that the time zone information can be read from.
- **filename** – This is an optional parameter specifying the source of the time zone information in the event that `fileobj` is a file object. If omitted and `fileobj` is a file stream, this parameter will be set either to `fileobj`'s `name` attribute or to `repr(fileobj)`.

See Sources for Time Zone and Daylight Saving Time Data for more information. Time zone files can be compiled from the IANA Time Zone database files with the zic time zone compiler

dst(*dt*)

fromutc(*dt*)

The `tzfile` implementation of `datetime.tzinfo.fromutc()`.

Parameters **dt** – A `datetime.datetime` object.

Raises

- **TypeError** – Raised if `dt` is not a `datetime.datetime` object.
- **ValueError** – Raised if this is called with a `dt` which does not have this `tzinfo` attached.

Returns Returns a `datetime.datetime` object representing the wall time in `self`'s time zone.

is_ambiguous(*dt*, *idx=None*)

Whether or not the "wall time" of a given datetime is ambiguous in this zone.

Parameters **dt** – A `datetime.datetime`, naive or time zone aware.

Returns Returns `True` if ambiguous, `False` otherwise.

New in version 2.6.0.

tzname(**args*, ***kwargs*)

utcoffset(*dt*)

class `dateutil.tz.`**tzrange**(*stdabbr*, *stdoffset=None*, *dstabbr=None*, *dstoffset=None*, *start=None*, *end=None*)

The `tzrange` object is a time zone specified by a set of offsets and abbreviations, equivalent to the way the `TZ` variable can be specified in POSIX-like systems, but using Python delta objects to specify DST start, end and offsets.

Parameters

- **stdabbr** – The abbreviation for standard time (e.g. `'EST'`).

- **stdoffset** – An integer or `datetime.timedelta` object or equivalent specifying the base offset from UTC.

  If unspecified, +00:00 is used.

- **dstabbr** – The abbreviation for DST / "Summer" time (e.g. `'EDT'`).

  If specified, with no other DST information, DST is assumed to occur and the default behavior or `dstoffset`, `start` and `end` is used. If unspecified and no other DST information is specified, it is assumed that this zone has no DST.

  If this is unspecified and other DST information is *is* specified, DST occurs in the zone but the time zone abbreviation is left unchanged.

- **dstoffset** – A an integer or `datetime.timedelta` object or equivalent specifying the UTC offset during DST. If unspecified and any other DST information is specified, it is assumed to be the STD offset +1 hour.

- **start** – A `relativedelta.relativedelta` object or equivalent specifying the time and time of year that daylight savings time starts. To specify, for example, that DST starts at 2AM on the 2nd Sunday in March, pass:

  ```
  relativedelta(hours=2, month=3, day=1, weekday=SU(+2))
  ```

  If unspecified and any other DST information is specified, the default value is 2 AM on the first Sunday in April.

- **end** – A `relativedelta.relativedelta` object or equivalent representing the time and time of year that daylight savings time ends, with the same specification method as in `start`. One note is that this should point to the first time in the *standard* zone, so if a transition occurs at 2AM in the DST zone and the clocks are set back 1 hour to 1AM, set the *hours* parameter to +1.

Examples:

```
>>> tzstr('EST5EDT') == tzrange("EST", -18000, "EDT")
True

>>> from dateutil.relativedelta import *
>>> range1 = tzrange("EST", -18000, "EDT")
>>> range2 = tzrange("EST", -18000, "EDT", -14400,
...                  relativedelta(hours=+2, month=4, day=1,
...                                weekday=SU(+1)),
...                  relativedelta(hours=+1, month=10, day=31,
...                                weekday=SU(-1)))
>>> tzstr('EST5EDT') == range1 == range2
True
```

transitions(*year*)

For a given year, get the DST on and off transition times, expressed always on the standard time side. For zones with no transitions, this function returns `None`.

Parameters **year** – The year whose transitions you would like to query.

Returns Returns a `tuple` of `datetime.datetime` objects, `(dston, dstoff)` for zones with an annual DST transition, or `None` for fixed offset zones.

class `dateutil.tz.`**tzstr**(*s*, *posix_offset=False*)

`tzstr` objects are time zone objects specified by a time-zone string as it would be passed to a `TZ` variable on POSIX-style systems (see the GNU C Library: TZ Variable for more details).

There is one notable exception, which is that POSIX-style time zones use an inverted offset format, so normally `GMT+3` would be parsed as an offset 3 hours *behind* GMT. The `tzstr` time zone object will parse this as an offset 3 hours *ahead* of GMT. If you would like to maintain the POSIX behavior, pass a `True` value to `posix_offset`.

The *tzrange* object provides the same functionality, but is specified using `relativedelta.relativedelta` objects. rather than strings.

Parameters

- **s** – A time zone string in `TZ` variable format. This can be a `bytes` (2.x: `str`), `str` (2.x: `unicode`) or a stream emitting unicode characters (e.g. `StringIO`).
- **posix_offset** – Optional. If set to `True`, interpret strings such as `GMT+3` or `UTC+3` as being 3 hours *behind* UTC rather than ahead, per the POSIX standard.

class `dateutil.tz.`**tzical**(*fileobj*)

This object is designed to parse an iCalendar-style `VTIMEZONE` structure as set out in RFC 2445 Section 4.6.5 into one or more *tzinfo* objects.

Parameters **fileobj** – A file or stream in iCalendar format, which should be UTF-8 encoded with CRLF endings.

get(*tzid=None*)

Retrieve a `datetime.tzinfo` object by its `tzid`.

Parameters **tzid** – If there is exactly one time zone available, omitting `tzid` or passing `None` value returns it. Otherwise a valid key (which can be retrieved from *keys()*) is required.

Raises **ValueError** – Raised if `tzid` is not specified but there are either more or fewer than 1 zone defined.

Returns Returns either a `datetime.tzinfo` object representing the relevant time zone or `None` if the `tzid` was not found.

keys()

Retrieves the available time zones as a list.

`dateutil.tz.`**gettz**(*name=None*)

`dateutil.tz.`**enfold**(*dt*, *fold=1*)

Provides a unified interface for assigning the `fold` attribute to datetimes both before and after the implementation of PEP-495.

Parameters **fold** – The value for the `fold` attribute in the returned datetime. This should be either 0 or 1.

Returns Returns an object for which `getattr(dt, 'fold', 0)` returns `fold` for all versions of Python. In versions prior to Python 3.6, this is a `_DatetimeWithFold` object, which is a subclass of `datetime.datetime` with the `fold` attribute added, if `fold` is 1.

New in version 2.6.0.

`dateutil.tz.`**datetime_ambiguous**(*dt*, *tz=None*)

Given a datetime and a time zone, determine whether or not a given datetime is ambiguous (i.e if there are two times differentiated only by their DST status).

Parameters

- **dt** – A `datetime.datetime` (whose time zone will be ignored if `tz` is provided.)
- **tz** – A `datetime.tzinfo` with support for the `fold` attribute. If `None` or not provided, the datetime's own time zone will be used.

Returns Returns a boolean value whether or not the "wall time" is ambiguous in `tz`.

New in version 2.6.0.

`dateutil.tz.`**`datetime_exists`**(*dt*, *tz=None*)

Given a datetime and a time zone, determine whether or not a given datetime would fall in a gap.

Parameters

- **`dt`** – A `datetime.datetime` (whose time zone will be ignored if `tz` is provided.)
- **`tz`** – A `datetime.tzinfo` with support for the `fold` attribute. If `None` or not provided, the datetime's own time zone will be used.

Returns Returns a boolean value whether or not the "wall time" exists in `tz`.

zoneinfo

`dateutil.zoneinfo.`**`get_zonefile_instance`**(*new_instance=False*)

This is a convenience function which provides a `ZoneInfoFile` instance using the data provided by the `dateutil` package. By default, it caches a single instance of the ZoneInfoFile object and returns that.

Parameters **`new_instance`** – If `True`, a new instance of `ZoneInfoFile` is instantiated and used as the cached instance for the next call. Otherwise, new instances are created only as necessary.

Returns Returns a `ZoneInfoFile` object.

New in version 2.6.

`dateutil.zoneinfo.`**`gettz`**(*name*)

This retrieves a time zone from the local zoneinfo tarball that is packaged with dateutil.

Parameters **`name`** – An IANA-style time zone name, as found in the zoneinfo file.

Returns Returns a *`dateutil.tz.tzfile`* time zone object.

> **Warning:** It is generally inadvisable to use this function, and it is only provided for API compatibility with earlier versions. This is *not* equivalent to `dateutil.tz.gettz()`, which selects an appropriate time zone based on the inputs, favoring system zoneinfo. This is ONLY for accessing the dateutil-specific zoneinfo (which may be out of date compared to the system zoneinfo).

Deprecated since version 2.6: If you need to use a specific zoneinfofile over the system zoneinfo, instantiate a `dateutil.zoneinfo.ZoneInfoFile` object and call `dateutil.zoneinfo.ZoneInfoFile.get(name)()` instead.

Use *`get_zonefile_instance()`* to retrieve an instance of the dateutil-provided zoneinfo.

`dateutil.zoneinfo.`**`gettz_db_metadata`**()

Get the zonefile metadata

See *zonefile_metadata*

Returns A dictionary with the database metadata

Deprecated since version 2.6: See deprecation warning in `zoneinfo.gettz()`. To get metadata, query the attribute `zoneinfo.ZoneInfoFile.metadata`.

zonefile_metadata

The zonefile metadata defines the version and exact location of the timezone database to download. It is used in the updatezinfo.py script. A json encoded file is included in the source-code, and within each tar file we produce. The json file is attached here:


```
{
    "metadata_version": 2.0,
    "releases_url": [
        "https://dateutil.github.io/tzdata/tzdata/",
        "ftp://ftp.iana.org/tz/releases/"
    ],
    "tzdata_file": "tzdata2017b.tar.gz",
    "tzdata_file_sha512":
↪"3e090dba1f52e4c63b4930b28f4bf38b56aabd6728f23094cb5801d10f4e464f17231f17b75b8866714bf98199c166ea84
↪",
    "tzversion": "2017b",
    "zonegroups": [
        "africa",
        "antarctica",
        "asia",
        "australasia",
        "europe",
        "northamerica",
        "southamerica",
        "pacificnew",
        "etcetera",
        "systemv",
        "factory",
        "backzone",
        "backward"
    ]
}
```


dateutil examples

Contents

- *tzrange examples*
- *tzfile examples*
- *tzical examples*
- *tzwin examples*
- *tzwinlocal examples*

relativedelta examples

Let's begin our trip:

```
>>> from datetime import *; from dateutil.relativedelta import *
>>> import calendar
```

Store some values:

```
>>> NOW = datetime.now()
>>> TODAY = date.today()
>>> NOW
datetime.datetime(2003, 9, 17, 20, 54, 47, 282310)
>>> TODAY
datetime.date(2003, 9, 17)
```

Next month

```
>>> NOW+relativedelta(months=+1)
datetime.datetime(2003, 10, 17, 20, 54, 47, 282310)
```

Next month, plus one week.

```
>>> NOW+relativedelta(months=+1, weeks=+1)
datetime.datetime(2003, 10, 24, 20, 54, 47, 282310)
```

Next month, plus one week, at 10am.

```
>>> TODAY+relativedelta(months=+1, weeks=+1, hour=10)
datetime.datetime(2003, 10, 24, 10, 0)
```

Here is another example using an absolute relativedelta. Notice the use of year and month (both singular) which causes the values to be *replaced* in the original datetime rather than performing an arithmetic operation on them.

```
>>> NOW+relativedelta(year=1, month=1)
datetime.datetime(1, 1, 17, 20, 54, 47, 282310)
```

Let's try the other way around. Notice that the hour setting we get in the relativedelta is relative, since it's a difference, and the weeks parameter has gone.

```
>>> relativedelta(datetime(2003, 10, 24, 10, 0), TODAY)
relativedelta(months=+1, days=+7, hours=+10)
```

One month before one year.

```
>>> NOW+relativedelta(years=+1, months=-1)
datetime.datetime(2004, 8, 17, 20, 54, 47, 282310)
```

How does it handle months with different numbers of days? Notice that adding one month will never cross the month boundary.

```
>>> date(2003,1,27)+relativedelta(months=+1)
datetime.date(2003, 2, 27)
>>> date(2003,1,31)+relativedelta(months=+1)
datetime.date(2003, 2, 28)
>>> date(2003,1,31)+relativedelta(months=+2)
datetime.date(2003, 3, 31)
```

The logic for years is the same, even on leap years.

```
>>> date(2000,2,28)+relativedelta(years=+1)
datetime.date(2001, 2, 28)
>>> date(2000,2,29)+relativedelta(years=+1)
datetime.date(2001, 2, 28)

>>> date(1999,2,28)+relativedelta(years=+1)
datetime.date(2000, 2, 28)
>>> date(1999,3,1)+relativedelta(years=+1)
datetime.date(2000, 3, 1)

>>> date(2001,2,28)+relativedelta(years=-1)
datetime.date(2000, 2, 28)
>>> date(2001,3,1)+relativedelta(years=-1)
datetime.date(2000, 3, 1)
```

Next friday

```
>>> TODAY+relativedelta(weekday=FR)
datetime.date(2003, 9, 19)

>>> TODAY+relativedelta(weekday=calendar.FRIDAY)
datetime.date(2003, 9, 19)
```

Last friday in this month.

```
>>> TODAY+relativedelta(day=31, weekday=FR(-1))
datetime.date(2003, 9, 26)
```

Next wednesday (it's today!).

```
>>> TODAY+relativedelta(weekday=WE(+1))
datetime.date(2003, 9, 17)
```

Next wednesday, but not today.

```
>>> TODAY+relativedelta(days=+1, weekday=WE(+1))
datetime.date(2003, 9, 24)
```

Following [http://www.cl.cam.ac.uk/~mgk25/iso-time.html ISO year week number notation] find the first day of the 15th week of 1997.

```
>>> datetime(1997,1,1)+relativedelta(day=4, weekday=MO(-1), weeks=+14)
datetime.datetime(1997, 4, 7, 0, 0)
```

How long ago has the millennium changed?

```
>>> relativedelta(NOW, date(2001,1,1))
relativedelta(years=+2, months=+8, days=+16,
              hours=+20, minutes=+54, seconds=+47, microseconds=+282310)
```

How old is John?

```
>>> johnbirthday = datetime(1978, 4, 5, 12, 0)
>>> relativedelta(NOW, johnbirthday)
relativedelta(years=+25, months=+5, days=+12,
          hours=+8, minutes=+54, seconds=+47, microseconds=+282310)
```

It works with dates too.

```
>>> relativedelta(TODAY, johnbirthday)
relativedelta(years=+25, months=+5, days=+11, hours=+12)
```

Obtain today's date using the yearday:

```
>>> date(2003, 1, 1)+relativedelta(yearday=260)
datetime.date(2003, 9, 17)
```

We can use today's date, since yearday should be absolute in the given year:

```
>>> TODAY+relativedelta(yearday=260)
datetime.date(2003, 9, 17)
```

Last year it should be in the same day:

```
>>> date(2002, 1, 1)+relativedelta(yearday=260)
datetime.date(2002, 9, 17)
```

But not in a leap year:

```
>>> date(2000, 1, 1)+relativedelta(yearday=260)
datetime.date(2000, 9, 16)
```

We can use the non-leap year day to ignore this:

```
>>> date(2000, 1, 1)+relativedelta(nlyearday=260)
datetime.date(2000, 9, 17)
```

rrule examples

These examples were converted from the RFC.

Prepare the environment.

```
>>> from dateutil.rrule import *
>>> from dateutil.parser import *
>>> from datetime import *

>>> import pprint
>>> import sys
>>> sys.displayhook = pprint.pprint
```

Daily, for 10 occurrences.

```
>>> list(rrule(DAILY, count=10,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 3, 9, 0),
 datetime.datetime(1997, 9, 4, 9, 0),
 datetime.datetime(1997, 9, 5, 9, 0),
 datetime.datetime(1997, 9, 6, 9, 0),
 datetime.datetime(1997, 9, 7, 9, 0),
 datetime.datetime(1997, 9, 8, 9, 0),
 datetime.datetime(1997, 9, 9, 9, 0),
 datetime.datetime(1997, 9, 10, 9, 0),
 datetime.datetime(1997, 9, 11, 9, 0)]
```

Daily until December 24, 1997

```
>>> list(rrule(DAILY,
...            dtstart=parse("19970902T090000"),
...            until=parse("19971224T000000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 3, 9, 0),
 datetime.datetime(1997, 9, 4, 9, 0),
 ...
 datetime.datetime(1997, 12, 21, 9, 0),
 datetime.datetime(1997, 12, 22, 9, 0),
 datetime.datetime(1997, 12, 23, 9, 0)]
```

Every other day, 5 occurrences.

```
>>> list(rrule(DAILY, interval=2, count=5,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 4, 9, 0),
 datetime.datetime(1997, 9, 6, 9, 0),
 datetime.datetime(1997, 9, 8, 9, 0),
 datetime.datetime(1997, 9, 10, 9, 0)]
```

Every 10 days, 5 occurrences.

```
>>> list(rrule(DAILY, interval=10, count=5,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 12, 9, 0),
 datetime.datetime(1997, 9, 22, 9, 0),
 datetime.datetime(1997, 10, 2, 9, 0),
 datetime.datetime(1997, 10, 12, 9, 0)]
```

Everyday in January, for 3 years.

```
>>> list(rrule(YEARLY, bymonth=1, byweekday=range(7),
...            dtstart=parse("19980101T090000"),
...            until=parse("20000131T090000")))
[datetime.datetime(1998, 1, 1, 9, 0),
 datetime.datetime(1998, 1, 2, 9, 0),
 ...
 datetime.datetime(1998, 1, 30, 9, 0),
 datetime.datetime(1998, 1, 31, 9, 0),
 datetime.datetime(1999, 1, 1, 9, 0),
 datetime.datetime(1999, 1, 2, 9, 0),
```

```
 ...
 datetime.datetime(1999, 1, 30, 9, 0),
 datetime.datetime(1999, 1, 31, 9, 0),
 datetime.datetime(2000, 1, 1, 9, 0),
 datetime.datetime(2000, 1, 2, 9, 0),
 ...
 datetime.datetime(2000, 1, 30, 9, 0),
 datetime.datetime(2000, 1, 31, 9, 0)]
```

Same thing, in another way.

```
>>> list(rrule(DAILY, bymonth=1,
...            dtstart=parse("19980101T090000"),
...            until=parse("20000131T090000")))
[datetime.datetime(1998, 1, 1, 9, 0),
 ...
 datetime.datetime(2000, 1, 31, 9, 0)]
```

Weekly for 10 occurrences.

```
>>> list(rrule(WEEKLY, count=10,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 9, 9, 0),
 datetime.datetime(1997, 9, 16, 9, 0),
 datetime.datetime(1997, 9, 23, 9, 0),
 datetime.datetime(1997, 9, 30, 9, 0),
 datetime.datetime(1997, 10, 7, 9, 0),
 datetime.datetime(1997, 10, 14, 9, 0),
 datetime.datetime(1997, 10, 21, 9, 0),
 datetime.datetime(1997, 10, 28, 9, 0),
 datetime.datetime(1997, 11, 4, 9, 0)]
```

Every other week, 6 occurrences.

```
>>> list(rrule(WEEKLY, interval=2, count=6,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 16, 9, 0),
 datetime.datetime(1997, 9, 30, 9, 0),
 datetime.datetime(1997, 10, 14, 9, 0),
 datetime.datetime(1997, 10, 28, 9, 0),
 datetime.datetime(1997, 11, 11, 9, 0)]
```

Weekly on Tuesday and Thursday for 5 weeks.

```
>>> list(rrule(WEEKLY, count=10, wkst=SU, byweekday=(TU,TH),
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 4, 9, 0),
 datetime.datetime(1997, 9, 9, 9, 0),
 datetime.datetime(1997, 9, 11, 9, 0),
 datetime.datetime(1997, 9, 16, 9, 0),
 datetime.datetime(1997, 9, 18, 9, 0),
 datetime.datetime(1997, 9, 23, 9, 0),
 datetime.datetime(1997, 9, 25, 9, 0),
 datetime.datetime(1997, 9, 30, 9, 0),
 datetime.datetime(1997, 10, 2, 9, 0)]
```

Every other week on Tuesday and Thursday, for 8 occurrences.

```
>>> list(rrule(WEEKLY, interval=2, count=8,
...            wkst=SU, byweekday=(TU,TH),
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 4, 9, 0),
 datetime.datetime(1997, 9, 16, 9, 0),
 datetime.datetime(1997, 9, 18, 9, 0),
 datetime.datetime(1997, 9, 30, 9, 0),
 datetime.datetime(1997, 10, 2, 9, 0),
 datetime.datetime(1997, 10, 14, 9, 0),
 datetime.datetime(1997, 10, 16, 9, 0)]
```

Monthly on the 1st Friday for ten occurrences.

```
>>> list(rrule(MONTHLY, count=10, byweekday=FR(1),
...            dtstart=parse("19970905T090000")))
[datetime.datetime(1997, 9, 5, 9, 0),
 datetime.datetime(1997, 10, 3, 9, 0),
 datetime.datetime(1997, 11, 7, 9, 0),
 datetime.datetime(1997, 12, 5, 9, 0),
 datetime.datetime(1998, 1, 2, 9, 0),
 datetime.datetime(1998, 2, 6, 9, 0),
 datetime.datetime(1998, 3, 6, 9, 0),
 datetime.datetime(1998, 4, 3, 9, 0),
 datetime.datetime(1998, 5, 1, 9, 0),
 datetime.datetime(1998, 6, 5, 9, 0)]
```

Every other month on the 1st and last Sunday of the month for 10 occurrences.

```
>>> list(rrule(MONTHLY, interval=2, count=10,
...            byweekday=(SU(1), SU(-1)),
...            dtstart=parse("19970907T090000")))
[datetime.datetime(1997, 9, 7, 9, 0),
 datetime.datetime(1997, 9, 28, 9, 0),
 datetime.datetime(1997, 11, 2, 9, 0),
 datetime.datetime(1997, 11, 30, 9, 0),
 datetime.datetime(1998, 1, 4, 9, 0),
 datetime.datetime(1998, 1, 25, 9, 0),
 datetime.datetime(1998, 3, 1, 9, 0),
 datetime.datetime(1998, 3, 29, 9, 0),
 datetime.datetime(1998, 5, 3, 9, 0),
 datetime.datetime(1998, 5, 31, 9, 0)]
```

Monthly on the second to last Monday of the month for 6 months.

```
>>> list(rrule(MONTHLY, count=6, byweekday=MO(-2),
...            dtstart=parse("19970922T090000")))
[datetime.datetime(1997, 9, 22, 9, 0),
 datetime.datetime(1997, 10, 20, 9, 0),
 datetime.datetime(1997, 11, 17, 9, 0),
 datetime.datetime(1997, 12, 22, 9, 0),
 datetime.datetime(1998, 1, 19, 9, 0),
 datetime.datetime(1998, 2, 16, 9, 0)]
```

Monthly on the third to the last day of the month, for 6 months.

```
>>> list(rrule(MONTHLY, count=6, bymonthday=-3,
...            dtstart=parse("19970928T090000")))
[datetime.datetime(1997, 9, 28, 9, 0),
 datetime.datetime(1997, 10, 29, 9, 0),
 datetime.datetime(1997, 11, 28, 9, 0),
 datetime.datetime(1997, 12, 29, 9, 0),
 datetime.datetime(1998, 1, 29, 9, 0),
 datetime.datetime(1998, 2, 26, 9, 0)]
```

Monthly on the 2nd and 15th of the month for 5 occurrences.

```
>>> list(rrule(MONTHLY, count=5, bymonthday=(2,15),
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 15, 9, 0),
 datetime.datetime(1997, 10, 2, 9, 0),
 datetime.datetime(1997, 10, 15, 9, 0),
 datetime.datetime(1997, 11, 2, 9, 0)]
```

Monthly on the first and last day of the month for 3 occurrences.

```
>>> list(rrule(MONTHLY, count=5, bymonthday=(-1,1,),
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 30, 9, 0),
 datetime.datetime(1997, 10, 1, 9, 0),
 datetime.datetime(1997, 10, 31, 9, 0),
 datetime.datetime(1997, 11, 1, 9, 0),
 datetime.datetime(1997, 11, 30, 9, 0)]
```

Every 18 months on the 10th thru 15th of the month for 10 occurrences.

```
>>> list(rrule(MONTHLY, interval=18, count=10,
...            bymonthday=range(10,16),
...            dtstart=parse("19970910T090000")))
[datetime.datetime(1997, 9, 10, 9, 0),
 datetime.datetime(1997, 9, 11, 9, 0),
 datetime.datetime(1997, 9, 12, 9, 0),
 datetime.datetime(1997, 9, 13, 9, 0),
 datetime.datetime(1997, 9, 14, 9, 0),
 datetime.datetime(1997, 9, 15, 9, 0),
 datetime.datetime(1999, 3, 10, 9, 0),
 datetime.datetime(1999, 3, 11, 9, 0),
 datetime.datetime(1999, 3, 12, 9, 0),
 datetime.datetime(1999, 3, 13, 9, 0)]
```

Every Tuesday, every other month, 6 occurences.

```
>>> list(rrule(MONTHLY, interval=2, count=6, byweekday=TU,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 9, 9, 0),
 datetime.datetime(1997, 9, 16, 9, 0),
 datetime.datetime(1997, 9, 23, 9, 0),
 datetime.datetime(1997, 9, 30, 9, 0),
 datetime.datetime(1997, 11, 4, 9, 0)]
```

Yearly in June and July for 10 occurrences.

```
>>> list(rrule(YEARLY, count=4, bymonth=(6,7),
...            dtstart=parse("19970610T090000")))
[datetime.datetime(1997, 6, 10, 9, 0),
 datetime.datetime(1997, 7, 10, 9, 0),
 datetime.datetime(1998, 6, 10, 9, 0),
 datetime.datetime(1998, 7, 10, 9, 0)]
```

Every 3rd year on the 1st, 100th and 200th day for 4 occurrences.

```
>>> list(rrule(YEARLY, count=4, interval=3, byyearday=(1,100,200),
...            dtstart=parse("19970101T090000")))
[datetime.datetime(1997, 1, 1, 9, 0),
 datetime.datetime(1997, 4, 10, 9, 0),
 datetime.datetime(1997, 7, 19, 9, 0),
 datetime.datetime(2000, 1, 1, 9, 0)]
```

Every 20th Monday of the year, 3 occurrences.

```
>>> list(rrule(YEARLY, count=3, byweekday=MO(20),
...            dtstart=parse("19970519T090000")))
[datetime.datetime(1997, 5, 19, 9, 0),
 datetime.datetime(1998, 5, 18, 9, 0),
 datetime.datetime(1999, 5, 17, 9, 0)]
```

Monday of week number 20 (where the default start of the week is Monday), 3 occurrences.

```
>>> list(rrule(YEARLY, count=3, byweekno=20, byweekday=MO,
...            dtstart=parse("19970512T090000")))
[datetime.datetime(1997, 5, 12, 9, 0),
 datetime.datetime(1998, 5, 11, 9, 0),
 datetime.datetime(1999, 5, 17, 9, 0)]
```

The week number 1 may be in the last year.

```
>>> list(rrule(WEEKLY, count=3, byweekno=1, byweekday=MO,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 12, 29, 9, 0),
 datetime.datetime(1999, 1, 4, 9, 0),
 datetime.datetime(2000, 1, 3, 9, 0)]
```

And the week numbers greater than 51 may be in the next year.

```
>>> list(rrule(WEEKLY, count=3, byweekno=52, byweekday=SU,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 12, 28, 9, 0),
 datetime.datetime(1998, 12, 27, 9, 0),
 datetime.datetime(2000, 1, 2, 9, 0)]
```

Only some years have week number 53:

```
>>> list(rrule(WEEKLY, count=3, byweekno=53, byweekday=MO,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1998, 12, 28, 9, 0),
 datetime.datetime(2004, 12, 27, 9, 0),
 datetime.datetime(2009, 12, 28, 9, 0)]
```

Every Friday the 13th, 4 occurrences.

```
>>> list(rrule(YEARLY, count=4, byweekday=FR, bymonthday=13,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1998, 2, 13, 9, 0),
 datetime.datetime(1998, 3, 13, 9, 0),
 datetime.datetime(1998, 11, 13, 9, 0),
 datetime.datetime(1999, 8, 13, 9, 0)]
```

Every four years, the first Tuesday after a Monday in November, 3 occurrences (U.S. Presidential Election day):

```
>>> list(rrule(YEARLY, interval=4, count=3, bymonth=11,
...            byweekday=TU, bymonthday=(2,3,4,5,6,7,8),
...            dtstart=parse("19961105T090000")))
[datetime.datetime(1996, 11, 5, 9, 0),
 datetime.datetime(2000, 11, 7, 9, 0),
 datetime.datetime(2004, 11, 2, 9, 0)]
```

The 3rd instance into the month of one of Tuesday, Wednesday or Thursday, for the next 3 months:

```
>>> list(rrule(MONTHLY, count=3, byweekday=(TU,WE,TH),
...            bysetpos=3, dtstart=parse("19970904T090000")))
[datetime.datetime(1997, 9, 4, 9, 0),
 datetime.datetime(1997, 10, 7, 9, 0),
 datetime.datetime(1997, 11, 6, 9, 0)]
```

The 2nd to last weekday of the month, 3 occurrences.

```
>>> list(rrule(MONTHLY, count=3, byweekday=(MO,TU,WE,TH,FR),
...            bysetpos=-2, dtstart=parse("19970929T090000")))
[datetime.datetime(1997, 9, 29, 9, 0),
 datetime.datetime(1997, 10, 30, 9, 0),
 datetime.datetime(1997, 11, 27, 9, 0)]
```

Every 3 hours from 9:00 AM to 5:00 PM on a specific day.

```
>>> list(rrule(HOURLY, interval=3,
...            dtstart=parse("19970902T090000"),
...            until=parse("19970902T170000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 2, 12, 0),
 datetime.datetime(1997, 9, 2, 15, 0)]
```

Every 15 minutes for 6 occurrences.

```
>>> list(rrule(MINUTELY, interval=15, count=6,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 2, 9, 15),
 datetime.datetime(1997, 9, 2, 9, 30),
 datetime.datetime(1997, 9, 2, 9, 45),
 datetime.datetime(1997, 9, 2, 10, 0),
 datetime.datetime(1997, 9, 2, 10, 15)]
```

Every hour and a half for 4 occurrences.

```
>>> list(rrule(MINUTELY, interval=90, count=4,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 2, 10, 30),
```

```
 datetime.datetime(1997, 9, 2, 12, 0),
 datetime.datetime(1997, 9, 2, 13, 30)]
```

Every 20 minutes from 9:00 AM to 4:40 PM for two days.

```
>>> list(rrule(MINUTELY, interval=20, count=48,
...            byhour=range(9,17), byminute=(0,20,40),
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 2, 9, 20),
 ...
 datetime.datetime(1997, 9, 2, 16, 20),
 datetime.datetime(1997, 9, 2, 16, 40),
 datetime.datetime(1997, 9, 3, 9, 0),
 datetime.datetime(1997, 9, 3, 9, 20),
 ...
 datetime.datetime(1997, 9, 3, 16, 20),
 datetime.datetime(1997, 9, 3, 16, 40)]
```

An example where the days generated makes a difference because of *wkst*.

```
>>> list(rrule(WEEKLY, interval=2, count=4,
...            byweekday=(TU,SU), wkst=MO,
...            dtstart=parse("19970805T090000")))
[datetime.datetime(1997, 8, 5, 9, 0),
 datetime.datetime(1997, 8, 10, 9, 0),
 datetime.datetime(1997, 8, 19, 9, 0),
 datetime.datetime(1997, 8, 24, 9, 0)]

>>> list(rrule(WEEKLY, interval=2, count=4,
...            byweekday=(TU,SU), wkst=SU,
...            dtstart=parse("19970805T090000")))
[datetime.datetime(1997, 8, 5, 9, 0),
 datetime.datetime(1997, 8, 17, 9, 0),
 datetime.datetime(1997, 8, 19, 9, 0),
 datetime.datetime(1997, 8, 31, 9, 0)]
```

rruleset examples

Daily, for 7 days, jumping Saturday and Sunday occurrences.

```
>>> set = rruleset()
>>> set.rrule(rrule(DAILY, count=7,
...                 dtstart=parse("19970902T090000")))
>>> set.exrule(rrule(YEARLY, byweekday=(SA,SU),
...                  dtstart=parse("19970902T090000")))
>>> list(set)
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 3, 9, 0),
 datetime.datetime(1997, 9, 4, 9, 0),
 datetime.datetime(1997, 9, 5, 9, 0),
 datetime.datetime(1997, 9, 8, 9, 0)]
```

Weekly, for 4 weeks, plus one time on day 7, and not on day 16.

```
>>> set = rruleset()
>>> set.rrule(rrule(WEEKLY, count=4,
...           dtstart=parse("19970902T090000")))
>>> set.rdate(datetime.datetime(1997, 9, 7, 9, 0))
>>> set.exdate(datetime.datetime(1997, 9, 16, 9, 0))
>>> list(set)
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 7, 9, 0),
 datetime.datetime(1997, 9, 9, 9, 0),
 datetime.datetime(1997, 9, 23, 9, 0)]
```

rrulestr() examples

Every 10 days, 5 occurrences.

```
>>> list(rrulestr("""
... DTSTART:19970902T090000
... RRULE:FREQ=DAILY;INTERVAL=10;COUNT=5
... """))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 12, 9, 0),
 datetime.datetime(1997, 9, 22, 9, 0),
 datetime.datetime(1997, 10, 2, 9, 0),
 datetime.datetime(1997, 10, 12, 9, 0)]
```

Same thing, but passing only the *RRULE* value.

```
>>> list(rrulestr("FREQ=DAILY;INTERVAL=10;COUNT=5",
...               dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 12, 9, 0),
 datetime.datetime(1997, 9, 22, 9, 0),
 datetime.datetime(1997, 10, 2, 9, 0),
 datetime.datetime(1997, 10, 12, 9, 0)]
```

Notice that when using a single rule, it returns an *rrule* instance, unless *forceset* was used.

```
>>> rrulestr("FREQ=DAILY;INTERVAL=10;COUNT=5")
<dateutil.rrule.rrule object at 0x...>

>>> rrulestr("""
... DTSTART:19970902T090000
... RRULE:FREQ=DAILY;INTERVAL=10;COUNT=5
... """)
<dateutil.rrule.rrule object at 0x...>

>>> rrulestr("FREQ=DAILY;INTERVAL=10;COUNT=5", forceset=True)
<dateutil.rrule.rruleset object at 0x...>
```

But when an *rruleset* is needed, it is automatically used.

```
>>> rrulestr("""
... DTSTART:19970902T090000
... RRULE:FREQ=DAILY;INTERVAL=10;COUNT=5
... RRULE:FREQ=DAILY;INTERVAL=5;COUNT=3
```

```
... """)
<dateutil.rrule.rruleset object at 0x...>
```

parse examples

The following code will prepare the environment:

```
>>> from dateutil.parser import *
>>> from dateutil.tz import *
>>> from datetime import *
>>> TZOFFSETS = {"BRST": -10800}
>>> BRSTTZ = tzoffset("BRST", -10800)
>>> DEFAULT = datetime(2003, 9, 25)
```

Some simple examples based on the *date* command, using the *ZOFFSET* dictionary to provide the BRST timezone offset.

```
>>> parse("Thu Sep 25 10:36:28 BRST 2003", tzinfos=TZOFFSETS)
datetime.datetime(2003, 9, 25, 10, 36, 28,
          tzinfo=tzoffset('BRST', -10800))

>>> parse("2003 10:36:28 BRST 25 Sep Thu", tzinfos=TZOFFSETS)
datetime.datetime(2003, 9, 25, 10, 36, 28,
          tzinfo=tzoffset('BRST', -10800))
```

Notice that since BRST is my local timezone, parsing it without further timezone settings will yield a *tzlocal* timezone.

```
>>> parse("Thu Sep 25 10:36:28 BRST 2003")
datetime.datetime(2003, 9, 25, 10, 36, 28, tzinfo=tzlocal())
```

We can also ask to ignore the timezone explicitly:

```
>>> parse("Thu Sep 25 10:36:28 BRST 2003", ignoretz=True)
datetime.datetime(2003, 9, 25, 10, 36, 28)
```

That's the same as processing a string without timezone:

```
>>> parse("Thu Sep 25 10:36:28 2003")
datetime.datetime(2003, 9, 25, 10, 36, 28)
```

Without the year, but passing our *DEFAULT* datetime to return the same year, no mattering what year we currently are in:

```
>>> parse("Thu Sep 25 10:36:28", default=DEFAULT)
datetime.datetime(2003, 9, 25, 10, 36, 28)
```

Strip it further:

```
>>> parse("Thu Sep 10:36:28", default=DEFAULT)
datetime.datetime(2003, 9, 25, 10, 36, 28)

>>> parse("Thu 10:36:28", default=DEFAULT)
datetime.datetime(2003, 9, 25, 10, 36, 28)

>>> parse("Thu 10:36", default=DEFAULT)
datetime.datetime(2003, 9, 25, 10, 36)
```

```
>>> parse("10:36", default=DEFAULT)
datetime.datetime(2003, 9, 25, 10, 36)
```

Strip in a different way:

```
>>> parse("Thu Sep 25 2003")
datetime.datetime(2003, 9, 25, 0, 0)

>>> parse("Sep 25 2003")
datetime.datetime(2003, 9, 25, 0, 0)

>>> parse("Sep 2003", default=DEFAULT)
datetime.datetime(2003, 9, 25, 0, 0)

>>> parse("Sep", default=DEFAULT)
datetime.datetime(2003, 9, 25, 0, 0)

>>> parse("2003", default=DEFAULT)
datetime.datetime(2003, 9, 25, 0, 0)
```

Another format, based on *date -R* (RFC822):

```
>>> parse("Thu, 25 Sep 2003 10:49:41 -0300")
datetime.datetime(2003, 9, 25, 10, 49, 41,
          tzinfo=tzoffset(None, -10800))
```

ISO format:

```
>>> parse("2003-09-25T10:49:41.5-03:00")
datetime.datetime(2003, 9, 25, 10, 49, 41, 500000,
          tzinfo=tzoffset(None, -10800))
```

Some variations:

```
>>> parse("2003-09-25T10:49:41")
datetime.datetime(2003, 9, 25, 10, 49, 41)

>>> parse("2003-09-25T10:49")
datetime.datetime(2003, 9, 25, 10, 49)

>>> parse("2003-09-25T10")
datetime.datetime(2003, 9, 25, 10, 0)

>>> parse("2003-09-25")
datetime.datetime(2003, 9, 25, 0, 0)
```

ISO format, without separators:

```
>>> parse("20030925T104941.5-0300")
datetime.datetime(2003, 9, 25, 10, 49, 41, 500000,
                  tzinfo=tzoffset(None, -10800))

>>> parse("20030925T104941-0300")
datetime.datetime(2003, 9, 25, 10, 49, 41,
          tzinfo=tzoffset(None, -10800))

>>> parse("20030925T104941")
```

```
datetime.datetime(2003, 9, 25, 10, 49, 41)

>>> parse("20030925T1049")
datetime.datetime(2003, 9, 25, 10, 49)

>>> parse("20030925T10")
datetime.datetime(2003, 9, 25, 10, 0)

>>> parse("20030925")
datetime.datetime(2003, 9, 25, 0, 0)
```

Everything together.

```
>>> parse("199709020900")
datetime.datetime(1997, 9, 2, 9, 0)
>>> parse("19970902090059")
datetime.datetime(1997, 9, 2, 9, 0, 59)
```

Different date orderings:

```
>>> parse("2003-09-25")
datetime.datetime(2003, 9, 25, 0, 0)

>>> parse("2003-Sep-25")
datetime.datetime(2003, 9, 25, 0, 0)

>>> parse("25-Sep-2003")
datetime.datetime(2003, 9, 25, 0, 0)

>>> parse("Sep-25-2003")
datetime.datetime(2003, 9, 25, 0, 0)

>>> parse("09-25-2003")
datetime.datetime(2003, 9, 25, 0, 0)

>>> parse("25-09-2003")
datetime.datetime(2003, 9, 25, 0, 0)
```

Check some ambiguous dates:

```
>>> parse("10-09-2003")
datetime.datetime(2003, 10, 9, 0, 0)

>>> parse("10-09-2003", dayfirst=True)
datetime.datetime(2003, 9, 10, 0, 0)

>>> parse("10-09-03")
datetime.datetime(2003, 10, 9, 0, 0)

>>> parse("10-09-03", yearfirst=True)
datetime.datetime(2010, 9, 3, 0, 0)
```

Other date separators are allowed:

```
>>> parse("2003.Sep.25")
datetime.datetime(2003, 9, 25, 0, 0)
```

```
>>> parse("2003/09/25")
datetime.datetime(2003, 9, 25, 0, 0)
```

Even with spaces:

```
>>> parse("2003 Sep 25")
datetime.datetime(2003, 9, 25, 0, 0)

>>> parse("2003 09 25")
datetime.datetime(2003, 9, 25, 0, 0)
```

Hours with letters work:

```
>>> parse("10h36m28.5s", default=DEFAULT)
datetime.datetime(2003, 9, 25, 10, 36, 28, 500000)

>>> parse("01s02h03m", default=DEFAULT)
datetime.datetime(2003, 9, 25, 2, 3, 1)

>>> parse("01h02m03", default=DEFAULT)
datetime.datetime(2003, 9, 25, 1, 2, 3)

>>> parse("01h02", default=DEFAULT)
datetime.datetime(2003, 9, 25, 1, 2)

>>> parse("01h02s", default=DEFAULT)
datetime.datetime(2003, 9, 25, 1, 0, 2)
```

With AM/PM:

```
>>> parse("10h am", default=DEFAULT)
datetime.datetime(2003, 9, 25, 10, 0)

>>> parse("10pm", default=DEFAULT)
datetime.datetime(2003, 9, 25, 22, 0)

>>> parse("12:00am", default=DEFAULT)
datetime.datetime(2003, 9, 25, 0, 0)

>>> parse("12pm", default=DEFAULT)
datetime.datetime(2003, 9, 25, 12, 0)
```

Some special treating for ''pertain" relations:

```
>>> parse("Sep 03", default=DEFAULT)
datetime.datetime(2003, 9, 3, 0, 0)

>>> parse("Sep of 03", default=DEFAULT)
datetime.datetime(2003, 9, 25, 0, 0)
```

Fuzzy parsing:

```
>>> s = "Today is 25 of September of 2003, exactly " \
...     "at 10:49:41 with timezone -03:00."
>>> parse(s, fuzzy=True)
datetime.datetime(2003, 9, 25, 10, 49, 41,
          tzinfo=tzoffset(None, -10800))
```

Other random formats:

```
>>> parse("Wed, July 10, '96")
datetime.datetime(1996, 7, 10, 0, 0)

>>> parse("1996.07.10 AD at 15:08:56 PDT", ignoretz=True)
datetime.datetime(1996, 7, 10, 15, 8, 56)

>>> parse("Tuesday, April 12, 1952 AD 3:30:42pm PST", ignoretz=True)
datetime.datetime(1952, 4, 12, 15, 30, 42)

>>> parse("November 5, 1994, 8:15:30 am EST", ignoretz=True)
datetime.datetime(1994, 11, 5, 8, 15, 30)

>>> parse("3rd of May 2001")
datetime.datetime(2001, 5, 3, 0, 0)

>>> parse("5:50 A.M. on June 13, 1990")
datetime.datetime(1990, 6, 13, 5, 50)
```

tzutc examples

```
>>> from datetime import *
>>> from dateutil.tz import *

>>> datetime.now()
datetime.datetime(2003, 9, 27, 9, 40, 1, 521290)

>>> datetime.now(tzutc())
datetime.datetime(2003, 9, 27, 12, 40, 12, 156379, tzinfo=tzutc())

>>> datetime.now(tzutc()).tzname()
'UTC'
```

tzoffset examples

```
>>> from datetime import *
>>> from dateutil.tz import *

>>> datetime.now(tzoffset("BRST", -10800))
datetime.datetime(2003, 9, 27, 9, 52, 43, 624904,
          tzinfo=tzinfo=tzoffset('BRST', -10800))

>>> datetime.now(tzoffset("BRST", -10800)).tzname()
'BRST'

>>> datetime.now(tzoffset("BRST", -10800)).astimezone(tzutc())
datetime.datetime(2003, 9, 27, 12, 53, 11, 446419,
          tzinfo=tzutc())
```

tzlocal examples

```
>>> from datetime import *
>>> from dateutil.tz import *

>>> datetime.now(tzlocal())
datetime.datetime(2003, 9, 27, 10, 1, 43, 673605,
          tzinfo=tzlocal())

>>> datetime.now(tzlocal()).tzname()
'BRST'

>>> datetime.now(tzlocal()).astimezone(tzoffset(None, 0))
datetime.datetime(2003, 9, 27, 13, 3, 0, 11493,
          tzinfo=tzoffset(None, 0))
```

tzstr examples

Here are examples of the recognized formats:

- *EST5EDT*
- *EST5EDT,4,0,6,7200,10,0,26,7200,3600*
- *EST5EDT,4,1,0,7200,10,-1,0,7200,3600*
- *EST5EDT4,M4.1.0/02:00:00,M10-5-0/02:00*
- *EST5EDT4,95/02:00:00,298/02:00*
- *EST5EDT4,J96/02:00:00,J299/02:00*

Notice that if daylight information is not present, but a daylight abbreviation was provided, *tzstr* will follow the convention of using the first sunday of April to start daylight saving, and the last sunday of October to end it. If start or end time is not present, 2AM will be used, and if the daylight offset is not present, the standard offset plus one hour will be used. This convention is the same as used in the GNU libc.

This also means that some of the above examples are exactly equivalent, and all of these examples are equivalent in the year of 2003.

Here is the example mentioned in the

[http://www.python.org/doc/current/lib/module-time.html time module documentation].

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

And here is an example showing the same information using *tzstr*, without touching system settings.

```
>>> tz1 = tzstr('EST+05EDT,M4.1.0,M10.5.0')
>>> tz2 = tzstr('AEST-10AEDT-11,M10.5.0,M3.5.0')
>>> dt = datetime(2003, 5, 8, 2, 7, 36, tzinfo=tz1)
>>> dt.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
```

```
>>> dt.astimezone(tz2).strftime('%X %x %Z')
'16:07:36 05/08/03 AEST'
```

Are these really equivalent?

```
>>> tzstr('EST5EDT') == tzstr('EST5EDT,4,1,0,7200,10,-1,0,7200,3600')
True
```

Check the daylight limit.

```
>>> tz = tzstr('EST+05EDT,M4.1.0,M10.5.0')
>>> datetime(2003, 4, 6, 1, 59, tzinfo=tz).tzname()
'EST'
>>> datetime(2003, 4, 6, 2, 00, tzinfo=tz).tzname()
'EDT'
>>> datetime(2003, 10, 26, 0, 59, tzinfo=tz).tzname()
'EDT'
>>> datetime(2003, 10, 26, 1, 00, tzinfo=tz).tzname()
'EST'
```

tzrange examples

```
>>> tzstr('EST5EDT') == tzrange("EST", -18000, "EDT")
True

>>> from dateutil.relativedelta import *
>>> range1 = tzrange("EST", -18000, "EDT")
>>> range2 = tzrange("EST", -18000, "EDT", -14400,
...                  relativedelta(hours=+2, month=4, day=1,
...                                weekday=SU(+1)),
...                  relativedelta(hours=+1, month=10, day=31,
...                                weekday=SU(-1)))
>>> tzstr('EST5EDT') == range1 == range2
True
```

Notice a minor detail in the last example: while the DST should end at 2AM, the delta will catch 1AM. That's because the daylight saving time should end at 2AM standard time (the difference between STD and DST is 1h in the given example) instead of the DST time. That's how the *tzinfo* subtypes should deal with the extra hour that happens when going back to the standard time. Check

[http://www.python.org/doc/current/lib/datetime-tzinfo.html tzinfo documentation]

for more information.

tzfile examples

```
>>> tz = tzfile("/etc/localtime")
>>> datetime.now(tz)
datetime.datetime(2003, 9, 27, 12, 3, 48, 392138,
          tzinfo=tzfile('/etc/localtime'))

>>> datetime.now(tz).astimezone(tzutc())
datetime.datetime(2003, 9, 27, 15, 3, 53, 70863,
          tzinfo=tzutc())
```

```
 >>> datetime.now(tz).tzname()
 'BRST'
 >>> datetime(2003, 1, 1, tzinfo=tz).tzname()
 'BRDT'
```

Check the daylight limit.

```
>>> tz = tzfile('/usr/share/zoneinfo/EST5EDT')
>>> datetime(2003, 4, 6, 1, 59, tzinfo=tz).tzname()
'EST'
>>> datetime(2003, 4, 6, 2, 00, tzinfo=tz).tzname()
'EDT'
>>> datetime(2003, 10, 26, 0, 59, tzinfo=tz).tzname()
'EDT'
>>> datetime(2003, 10, 26, 1, 00, tzinfo=tz).tzname()
'EST'
```

tzical examples

Here is a sample file extracted from the RFC. This file defines the *EST5EDT* timezone, and will be used in the following example.

```
BEGIN:VTIMEZONE
TZID:US-Eastern
LAST-MODIFIED:19870101T000000Z
TZURL:http://zones.stds_r_us.net/tz/US-Eastern
BEGIN:STANDARD
DTSTART:19671029T020000
RRULE:FREQ=YEARLY;BYDAY=-1SU;BYMONTH=10
TZOFFSETFROM:-0400
TZOFFSETTO:-0500
TZNAME:EST
END:STANDARD
BEGIN:DAYLIGHT
DTSTART:19870405T020000
RRULE:FREQ=YEARLY;BYDAY=1SU;BYMONTH=4
TZOFFSETFROM:-0500
TZOFFSETTO:-0400
TZNAME:EDT
END:DAYLIGHT
END:VTIMEZONE
```

And here is an example exploring a *tzical* type:

```
>>> from dateutil.tz import *; from datetime import *

>>> tz = tzical('samples/EST5EDT.ics')
>>> tz.keys()
['US-Eastern']

>>> est = tz.get('US-Eastern')
>>> est
<tzicalvtz 'US-Eastern'>

>>> datetime.now(est)
datetime.datetime(2003, 10, 6, 19, 44, 18, 667987,
```

```
          tzinfo=<tzicalvtz 'US-Eastern'>)

>>> est == tz.get()
True
```

Let's check the daylight ranges, as usual:

```
>>> datetime(2003, 4, 6, 1, 59, tzinfo=est).tzname()
'EST'
>>> datetime(2003, 4, 6, 2, 00, tzinfo=est).tzname()
'EDT'

>>> datetime(2003, 10, 26, 0, 59, tzinfo=est).tzname()
'EDT'
>>> datetime(2003, 10, 26, 1, 00, tzinfo=est).tzname()
'EST'
```

tzwin examples

```
>>> tz = tzwin("E. South America Standard Time")
```

tzwinlocal examples

```
>>> tz = tzwinlocal()
```

vim:ts=4:sw=4:et

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

T

U

V

W