

---

# **DateParser Documentation**

***Release 1.2.1***

**Scrapinghub**

**May 27, 2025**



# CONTENTS

<b>1</b>	<b>Documentation</b>	<b>3</b>
<b>2</b>	<b>Introduction to dateparser</b>	<b>5</b>
2.1	Features . . . . .	5
2.2	Basic Usage . . . . .	5
2.3	Advanced Usage . . . . .	10
2.4	Dependencies . . . . .	10
2.5	Supported languages and locales . . . . .	11
2.6	Supported Calendars . . . . .	11
2.7	Indices and tables . . . . .	11
	<b>Python Module Index</b>	<b>53</b>
	<b>Index</b>	<b>55</b>



*dateparser* provides modules to easily parse localized dates in almost any string formats commonly found on web pages.



## DOCUMENTATION

This documentation is built automatically and can be found on [Read the Docs](#).



## INTRODUCTION TO DATEPARSER

### 2.1 Features

- Generic parsing of dates in over 200 language locales plus numerous formats in a language agnostic fashion.
- Generic parsing of relative dates like: '1 min ago', '2 weeks ago', '3 months, 1 week and 1 day ago', 'in 2 days', 'tomorrow'.
- Generic parsing of dates with time zones abbreviations or UTC offsets like: 'August 14, 2015 EST', 'July 4, 2013 PST', '21 July 2013 10:15 pm +0500'.
- Date lookup in longer texts.
- Support for non-Gregorian calendar systems. See [Supported Calendars](#).
- Extensive test coverage.

### 2.2 Basic Usage

The most straightforward way is to use the `dateparser.parse` function, that wraps around most of the functionality in the module.

```
dateparser.parse(date_string, date_formats=None, languages=None, locales=None, region=None, settings=None, detect_languages_function=None)
```

Parse date and time from given date string.

#### Parameters

- **date\_string** (*str*) – A string representing date and/or time in a recognizably valid format.
- **date\_formats** (*list*) – A list of format strings using directives as given [here](#). The parser applies formats one by one, taking into account the detected languages/locales.
- **languages** (*list*) – A list of language codes, e.g. ['en', 'es', 'zh-Hant']. If locales are not given, languages and region are used to construct locales for translation.
- **locales** (*list*) – A list of locale codes, e.g. ['fr-PF', 'qu-EC', 'af-NA']. The parser uses only these locales to translate date string.
- **region** (*str*) – A region code, e.g. 'IN', '001', 'NE'. If locales are not given, languages and region are used to construct locales for translation.
- **settings** (*dict*) – Configure customized behavior using settings defined in `dateparser.conf.Settings`.
- **detect\_languages\_function** (*function*) – A function for language detection that takes as input a string (the *date\_string*) and a *confidence\_threshold*, and returns a list of detected language codes. Note: this function is only used if `languages` and `locales` are not provided.

**Returns**

Returns `datetime` representing parsed date if successful, else returns `None`

**Return type**

`datetime`.

**Raises**

`ValueError: Unknown Language`, `TypeError: Languages argument must be a list`, `Setting-ValidationError: A provided setting is not valid`.

## 2.2.1 Popular Formats

```
>>> import dateparser
>>> dateparser.parse('12/12/12')
datetime.datetime(2012, 12, 12, 0, 0)
>>> dateparser.parse('Fri, 12 Dec 2014 10:55:50')
datetime.datetime(2014, 12, 12, 10, 55, 50)
>>> dateparser.parse('Martes 21 de Octubre de 2014') # Spanish (Tuesday 21 October
↳2014)
datetime.datetime(2014, 10, 21, 0, 0)
>>> dateparser.parse('Le 11 Décembre 2014 à 09:00') # French (11 December 2014 at
↳09:00)
datetime.datetime(2014, 12, 11, 9, 0)
>>> dateparser.parse('13 января 2015 г. в 13:34') # Russian (13 January 2015 at
↳13:34)
datetime.datetime(2015, 1, 13, 13, 34)
>>> dateparser.parse('1 ตุลาคม 2005, 1:00 AM') # Thai (1 October 2005, 1:00 AM)
datetime.datetime(2005, 10, 1, 1, 0)
```

This will try to parse a date from the given string, attempting to detect the language each time.

You can specify the language(s), if known, using `languages` argument. In this case, given languages are used and language detection is skipped:

```
>>> dateparser.parse('2015, Ago 15, 1:08 pm', languages=['pt', 'es'])
datetime.datetime(2015, 8, 15, 13, 8)
```

If you know the possible formats of the dates, you can use the `date_formats` argument:

```
>>> dateparser.parse('22 Décembre 2010', date_formats=['%d %B %Y'])
datetime.datetime(2010, 12, 22, 0, 0)
```

## 2.2.2 Relative Dates

```
>>> parse('1 hour ago')
datetime.datetime(2015, 5, 31, 23, 0)
>>> parse('Il ya 2 heures') # French (2 hours ago)
datetime.datetime(2015, 5, 31, 22, 0)
>>> parse('1 anno 2 mesi') # Italian (1 year 2 months)
datetime.datetime(2014, 4, 1, 0, 0)
>>> parse('yaklaşık 23 saat önce') # Turkish (23 hours ago)
datetime.datetime(2015, 5, 31, 1, 0)
>>> parse('Hace una semana') # Spanish (a week ago)
datetime.datetime(2015, 5, 25, 0, 0)
```

(continues on next page)

(continued from previous page)

```
>>> parse('2小时前') # Chinese (2 hours ago)
datetime.datetime(2015, 5, 31, 22, 0)
```

**Note**

Testing above code might return different values for you depending on your environment's current date and time.

**Note**

For *Finnish* language, please specify `settings={'SKIP_TOKENS': []}` to correctly parse relative dates.

## 2.2.3 OOTB Language Based Date Order Preference

```
>>> # parsing ambiguous date
>>> parse('02-03-2016') # assumes english language, uses MDY date order
datetime.datetime(2016, 2, 3, 0, 0)
>>> parse('le 02-03-2016') # detects french, uses DMY date order
datetime.datetime(2016, 3, 2, 0, 0)
```

**Note**

Ordering is not locale based, that's why do not expect *DMY* order for UK/Australia English. You can specify date order in that case as follows using *Settings*:

```
>>> parse('18-12-15 06:00', settings={'DATE_ORDER': 'DMY'})
datetime.datetime(2015, 12, 18, 6, 0)
```

For more on date order, please look at *Settings*.

## 2.2.4 Timezone and UTC Offset

By default, *dateparser* returns tzaware *datetime* if timezone is present in date string. Otherwise, it returns a naive *datetime* object.

```
>>> parse('January 12, 2012 10:00 PM EST')
datetime.datetime(2012, 1, 12, 22, 0, tzinfo=<StaticTzInfo 'EST'>)
```

```
>>> parse('January 12, 2012 10:00 PM -0500')
datetime.datetime(2012, 1, 12, 22, 0, tzinfo=<StaticTzInfo 'UTC\ -05:00'>)
```

```
>>> parse('2 hours ago EST')
datetime.datetime(2017, 3, 10, 15, 55, 39, 579667, tzinfo=<StaticTzInfo 'EST'
↳ '>')
```

```
>>> parse('2 hours ago -0500')
datetime.datetime(2017, 3, 10, 15, 59, 30, 193431, tzinfo=<StaticTzInfo 'UTC\
↳ -05:00'>)
```

If date has no timezone name/abbreviation or offset, you can specify it using *TIMEZONE* setting.

```
>>> parse('January 12, 2012 10:00 PM', settings={'TIMEZONE': 'US/Eastern'})
datetime.datetime(2012, 1, 12, 22, 0)
```

```
>>> parse('January 12, 2012 10:00 PM', settings={'TIMEZONE': '+0500'})
datetime.datetime(2012, 1, 12, 22, 0)
```

*TIMEZONE* option may not be useful alone as it only attaches given timezone to resultant *datetime* object. But can be useful in cases where you want conversions from and to different timezones or when simply want a tzaware date with given timezone info attached.

```
>>> parse('January 12, 2012 10:00 PM', settings={'TIMEZONE': 'US/Eastern', 'RETURN_AS_
↳TIMEZONE_AWARE': True})
datetime.datetime(2012, 1, 12, 22, 0, tzinfo=<DstTzInfo 'US/Eastern' EST-1 day,
↳19:00:00 STD>)
```

```
>>> parse('10:00 am', settings={'TIMEZONE': 'EST', 'TO_TIMEZONE': 'EDT'})
datetime.datetime(2016, 9, 25, 11, 0)
```

Some more use cases for conversion of timezones.

```
>>> parse('10:00 am EST', settings={'TO_TIMEZONE': 'EDT'}) # date string has
↳timezone info
datetime.datetime(2017, 3, 12, 11, 0, tzinfo=<StaticTzInfo 'EDT'>)
```

```
>>> parse('now EST', settings={'TO_TIMEZONE': 'UTC'}) # relative dates
datetime.datetime(2017, 3, 10, 23, 24, 47, 371823, tzinfo=<StaticTzInfo 'UTC'>)
```

In case, no timezone is present in date string or defined in *Settings*. You can still return tzaware *datetime*. It is especially useful in case of relative dates when uncertain what timezone is relative base.

```
>>> parse('2 minutes ago', settings={'RETURN_AS_TIMEZONE_AWARE': True})
datetime.datetime(2017, 3, 11, 4, 25, 24, 152670, tzinfo=<DstTzInfo 'Asia/Karachi'
↳PKT+5:00:00 STD>)
```

In case, you want to compute relative dates in UTC instead of default system's local timezone, you can use *TIMEZONE* setting.

```
>>> parse('4 minutes ago', settings={'TIMEZONE': 'UTC'})
datetime.datetime(2017, 3, 10, 23, 27, 59, 647248, tzinfo=<StaticTzInfo 'UTC'>)
```

### Note

In case, when timezone is present both in string and also specified using *Settings*, string is parsed into tzaware representation and then converted to timezone specified in *Settings*.

```
>>> parse('10:40 pm PKT', settings={'TIMEZONE': 'UTC'})
datetime.datetime(2017, 3, 12, 17, 40, tzinfo=<StaticTzInfo 'UTC'>)
```

```
>>> parse('20 mins ago EST', settings={'TIMEZONE': 'UTC'})
datetime.datetime(2017, 3, 12, 21, 16, 0, 885091, tzinfo=<StaticTzInfo 'UTC'>)
```

For more on timezones, please look at *Settings*.

## 2.2.5 Incomplete Dates

```
>>> from dateparser import parse
>>> parse('December 2015') # default behavior
datetime.datetime(2015, 12, 16, 0, 0)
>>> parse('December 2015', settings={'PREFER_DAY_OF_MONTH': 'last'})
datetime.datetime(2015, 12, 31, 0, 0)
>>> parse('December 2015', settings={'PREFER_DAY_OF_MONTH': 'first'})
datetime.datetime(2015, 12, 1, 0, 0)
```

```
>>> parse('March')
datetime.datetime(2015, 3, 16, 0, 0)
>>> parse('March', settings={'PREFER_DATES_FROM': 'future'})
datetime.datetime(2016, 3, 16, 0, 0)
>>> # parsing with preference set for 'past'
>>> parse('August', settings={'PREFER_DATES_FROM': 'past'})
datetime.datetime(2015, 8, 15, 0, 0)
```

```
>>> import dateparser
>>> dateparser.parse("2015") # default behavior
datetime.datetime(2015, 3, 27, 0, 0)
>>> dateparser.parse("2015", settings={"PREFER_MONTH_OF_YEAR": "last"})
datetime.datetime(2015, 12, 27, 0, 0)
>>> dateparser.parse("2015", settings={"PREFER_MONTH_OF_YEAR": "first"})
datetime.datetime(2015, 1, 27, 0, 0)
>>> dateparser.parse("2015", settings={"PREFER_MONTH_OF_YEAR": "current"})
datetime.datetime(2015, 3, 27, 0, 0)
```

You can also ignore parsing incomplete dates altogether by setting *STRICT\_PARSING* flag as follows:

```
>>> parse('December 2015', settings={'STRICT_PARSING': True})
None
```

For more on handling incomplete dates, please look at *Settings*.

## 2.2.6 Search for Dates in Longer Chunks of Text

### Warning

Support for searching dates is really limited and needs a lot of improvement, we look forward to community's contribution to get better on that part. See "*Contributing*".

You can extract dates from longer strings of text. They are returned as list of tuples with text chunk containing the date and parsed datetime object.

```
dateparser.search.search_dates(text, languages=None, settings=None, add_detected_language=False,
                                detect_languages_function=None)
```

Find all substrings of the given string which represent date and/or time and parse them.

### Parameters

- **text** (*str*) – A string in a natural language which may contain date and/or time expressions.
- **languages** (*list*) – A list of two letters language codes.e.g. ['en', 'es']. If languages are given, it will not attempt to detect the language.
- **settings** (*dict*) – Configure customized behavior using settings defined in `dateparser.conf.Settings`.
- **add\_detected\_language** (*bool*) – Indicates if we want the detected language returned in the tuple.
- **detect\_languages\_function** (*function*) – A function for language detection that takes as input a *text* and a *confidence\_threshold*, and returns a list of detected language codes. Note: `detect_languages_function` is only uses if *languages* are not provided.

**Returns**

Returns list of tuples containing: substrings representing date and/or time, corresponding `datetime.datetime` object and detected language if `add_detected_language` is True. Returns None if no dates that can be parsed are found.

**Return type**

list

**Raises**

`ValueError` - Unknown Language

```
>>> from dateparser.search import search_dates
>>> search_dates('The first artificial Earth satellite was launched on 4 October_
↳1957.')
[('on 4 October 1957', datetime.datetime(1957, 10, 4, 0, 0))]
```

```
>>> search_dates('The first artificial Earth satellite was launched on 4 October_
↳1957.',
>>>                  add_detected_language=True)
[('on 4 October 1957', datetime.datetime(1957, 10, 4, 0, 0), 'en')]
```

```
>>> search_dates("The client arrived to the office for the first time in March_
↳3rd, 2004 "
>>>                  "and got serviced, after a couple of months, on May 6th 2004,_"
↳the customer "
>>>                  "returned indicating a defect on the part")
[('in March 3rd, 2004 and', datetime.datetime(2004, 3, 3, 0, 0)),
 ('on May 6th 2004', datetime.datetime(2004, 5, 6, 0, 0))]
```

## 2.3 Advanced Usage

If you need more control over what is being parser check the [Settings](#) section as well as the [Using DateDataParser](#) section.

## 2.4 Dependencies

`dateparser` relies on following libraries in some ways:

- `dateutil`'s module `relativedelta` for its freshness parser.
- `convertdate` to convert *Jalali* dates to *Gregorian*.
- `hijridate` to convert *Hijri* dates to *Gregorian*.

- `tzlocal` to reliably get local timezone.
- `ruamel.yaml` (optional) for operations on language files.

## 2.5 Supported languages and locales

You can check the supported locales by visiting the “*Supported languages and locales*” section.

## 2.6 Supported Calendars

Apart from the Georgian calendar, *dateparser* supports the *Persian Jalali calendar* and the *Hijri/Islami calendar*

To be able to use them you need to install the *calendar* extra by typing:

```
pip install dateparser[calendars]
```

- Example using the *Persian Jalali calendar*. For more information, refer to [Persian Jalali Calendar](#).

```
>>> from dateparser.calendars.jalali import JalaliCalendar
>>> JalaliCalendar('۱۳۸۷ اسفند ۱۳ م سی و چهارم').get_date()
DateData(date_obj=datetime.datetime(2009, 3, 20, 0, 0), period='day', locale=None)
```

- Example using the *Hijri/Islamic Calendar*. For more information, refer to [Hijri Calendar](#).

```
>>> from dateparser.calendars.hijri import HijriCalendar
>>> HijriCalendar('17-01-1437 هـ 08:30 مساءً').get_date()
DateData(date_obj=datetime.datetime(2015, 10, 30, 20, 30), period='day',
↪ locale=None)
```

### Note

*HijriCalendar* only works with Python  $\geq 3.7$ .

## 2.7 Indices and tables

Contents:

### 2.7.1 Introduction to dateparser

#### Features

- Generic parsing of dates in over 200 language locales plus numerous formats in a language agnostic fashion.
- Generic parsing of relative dates like: '1 min ago', '2 weeks ago', '3 months, 1 week and 1 day ago', 'in 2 days', 'tomorrow'.
- Generic parsing of dates with time zones abbreviations or UTC offsets like: 'August 14, 2015 EST', 'July 4, 2013 PST', '21 July 2013 10:15 pm +0500'.
- Date lookup in longer texts.
- Support for non-Gregorian calendar systems. See *Supported Calendars*.
- Extensive test coverage.

## Basic Usage

The most straightforward way is to use the `dateparser.parse` function, that wraps around most of the functionality in the module.

```
dateparser.parse(date_string, date_formats=None, languages=None, locales=None, region=None, settings=None,
                  detect_languages_function=None)
```

Parse date and time from given date string.

### Parameters

- **date\_string** (*str*) – A string representing date and/or time in a recognizably valid format.
- **date\_formats** (*list*) – A list of format strings using directives as given [here](#). The parser applies formats one by one, taking into account the detected languages/locales.
- **languages** (*list*) – A list of language codes, e.g. ['en', 'es', 'zh-Hant']. If locales are not given, languages and region are used to construct locales for translation.
- **locales** (*list*) – A list of locale codes, e.g. ['fr-PF', 'qu-EC', 'af-NA']. The parser uses only these locales to translate date string.
- **region** (*str*) – A region code, e.g. 'IN', '001', 'NE'. If locales are not given, languages and region are used to construct locales for translation.
- **settings** (*dict*) – Configure customized behavior using settings defined in `dateparser.conf.Settings`.
- **detect\_languages\_function** (*function*) – A function for language detection that takes as input a string (the *date\_string*) and a *confidence\_threshold*, and returns a list of detected language codes. Note: this function is only used if *languages* and *locales* are not provided.

### Returns

Returns `datetime` representing parsed date if successful, else returns `None`

### Return type

`datetime`.

### Raises

`ValueError: Unknown Language`, `TypeError: Languages argument must be a list`, `Setting-ValidationError: A provided setting is not valid`.

## Popular Formats

```
>>> import dateparser
>>> dateparser.parse('12/12/12')
datetime.datetime(2012, 12, 12, 0, 0)
>>> dateparser.parse('Fri, 12 Dec 2014 10:55:50')
datetime.datetime(2014, 12, 12, 10, 55, 50)
>>> dateparser.parse('Martes 21 de Octubre de 2014') # Spanish (Tuesday 21 October
↳ 2014)
datetime.datetime(2014, 10, 21, 0, 0)
>>> dateparser.parse('Le 11 Décembre 2014 à 09:00') # French (11 December 2014 at
↳ 09:00)
datetime.datetime(2014, 12, 11, 9, 0)
>>> dateparser.parse('13 января 2015 г. в 13:34') # Russian (13 January 2015 at
↳ 13:34)
datetime.datetime(2015, 1, 13, 13, 34)
```

(continues on next page)

(continued from previous page)

```
>>> dateparser.parse('1 ตุลาคม 2005, 1:00 AM') # Thai (1 October 2005, 1:00 AM)
datetime.datetime(2005, 10, 1, 1, 0)
```

This will try to parse a date from the given string, attempting to detect the language each time.

You can specify the language(s), if known, using `languages` argument. In this case, given languages are used and language detection is skipped:

```
>>> dateparser.parse('2015, Ago 15, 1:08 pm', languages=['pt', 'es'])
datetime.datetime(2015, 8, 15, 13, 8)
```

If you know the possible formats of the dates, you can use the `date_formats` argument:

```
>>> dateparser.parse('22 Décembre 2010', date_formats=['%d %B %Y'])
datetime.datetime(2010, 12, 22, 0, 0)
```

## Relative Dates

```
>>> parse('1 hour ago')
datetime.datetime(2015, 5, 31, 23, 0)
>>> parse('Il ya 2 heures') # French (2 hours ago)
datetime.datetime(2015, 5, 31, 22, 0)
>>> parse('1 anno 2 mesi') # Italian (1 year 2 months)
datetime.datetime(2014, 4, 1, 0, 0)
>>> parse('yaklaşık 23 saat önce') # Turkish (23 hours ago)
datetime.datetime(2015, 5, 31, 1, 0)
>>> parse('Hace una semana') # Spanish (a week ago)
datetime.datetime(2015, 5, 25, 0, 0)
>>> parse('2小时前') # Chinese (2 hours ago)
datetime.datetime(2015, 5, 31, 22, 0)
```

### Note

Testing above code might return different values for you depending on your environment's current date and time.

### Note

For *Finnish* language, please specify `settings={'SKIP_TOKENS': []}` to correctly parse relative dates.

## OOTB Language Based Date Order Preference

```
>>> # parsing ambiguous date
>>> parse('02-03-2016') # assumes english language, uses MDY date order
datetime.datetime(2016, 2, 3, 0, 0)
>>> parse('le 02-03-2016') # detects french, uses DMY date order
datetime.datetime(2016, 3, 2, 0, 0)
```

### Note

Ordering is not locale based, that's why do not expect *DMY* order for UK/Australia English. You can specify date order in that case as follows using *Settings*:

```
>>> parse('18-12-15 06:00', settings={'DATE_ORDER': 'DMY'})
datetime.datetime(2015, 12, 18, 6, 0)
```

For more on date order, please look at *Settings*.

### Timezone and UTC Offset

By default, *dateparser* returns tzaware *datetime* if timezone is present in date string. Otherwise, it returns a naive *datetime* object.

```
>>> parse('January 12, 2012 10:00 PM EST')
datetime.datetime(2012, 1, 12, 22, 0, tzinfo=<StaticTzInfo 'EST'>)
```

```
>>> parse('January 12, 2012 10:00 PM -0500')
datetime.datetime(2012, 1, 12, 22, 0, tzinfo=<StaticTzInfo 'UTC\ -05:00'>)
```

```
>>> parse('2 hours ago EST')
datetime.datetime(2017, 3, 10, 15, 55, 39, 579667, tzinfo=<StaticTzInfo 'EST'>
↳ '>')
```

```
>>> parse('2 hours ago -0500')
datetime.datetime(2017, 3, 10, 15, 59, 30, 193431, tzinfo=<StaticTzInfo 'UTC\ 
↳ -05:00'>)
```

If date has no timezone name/abbreviation or offset, you can specify it using *TIMEZONE* setting.

```
>>> parse('January 12, 2012 10:00 PM', settings={'TIMEZONE': 'US/Eastern'})
datetime.datetime(2012, 1, 12, 22, 0)
```

```
>>> parse('January 12, 2012 10:00 PM', settings={'TIMEZONE': '+0500'})
datetime.datetime(2012, 1, 12, 22, 0)
```

*TIMEZONE* option may not be useful alone as it only attaches given timezone to resultant *datetime* object. But can be useful in cases where you want conversions from and to different timezones or when simply want a tzaware date with given timezone info attached.

```
>>> parse('January 12, 2012 10:00 PM', settings={'TIMEZONE': 'US/Eastern', 'RETURN_AS_
↳ TIMEZONE_AWARE': True})
datetime.datetime(2012, 1, 12, 22, 0, tzinfo=<DstTzInfo 'US/Eastern' EST-1 day, 
↳ 19:00:00 STD>)
```

```
>>> parse('10:00 am', settings={'TIMEZONE': 'EST', 'TO_TIMEZONE': 'EDT'})
datetime.datetime(2016, 9, 25, 11, 0)
```

Some more use cases for conversion of timezones.

```
>>> parse('10:00 am EST', settings={'TO_TIMEZONE': 'EDT'}) # date string has
↳ timezone info
datetime.datetime(2017, 3, 12, 11, 0, tzinfo=<StaticTzInfo 'EDT'>)
```

```
>>> parse('now EST', settings={'TO_TIMEZONE': 'UTC'}) # relative dates
datetime.datetime(2017, 3, 10, 23, 24, 47, 371823, tzinfo=<StaticTzInfo 'UTC'>)
```

In case, no timezone is present in date string or defined in *Settings*. You can still return tzaware datetime. It is especially useful in case of relative dates when uncertain what timezone is relative base.

```
>>> parse('2 minutes ago', settings={'RETURN_AS_TIMEZONE_AWARE': True})
datetime.datetime(2017, 3, 11, 4, 25, 24, 152670, tzinfo=<DstTzInfo 'Asia/Karachi'
↳ PKT+5:00:00 STD>)
```

In case, you want to compute relative dates in UTC instead of default system's local timezone, you can use *TIMEZONE* setting.

```
>>> parse('4 minutes ago', settings={'TIMEZONE': 'UTC'})
datetime.datetime(2017, 3, 10, 23, 27, 59, 647248, tzinfo=<StaticTzInfo 'UTC'>)
```

#### Note

In case, when timezone is present both in string and also specified using *Settings*, string is parsed into tzaware representation and then converted to timezone specified in *Settings*.

```
>>> parse('10:40 pm PKT', settings={'TIMEZONE': 'UTC'})
datetime.datetime(2017, 3, 12, 17, 40, tzinfo=<StaticTzInfo 'UTC'>)
```

```
>>> parse('20 mins ago EST', settings={'TIMEZONE': 'UTC'})
datetime.datetime(2017, 3, 12, 21, 16, 0, 885091, tzinfo=<StaticTzInfo 'UTC'>)
```

For more on timezones, please look at *Settings*.

## Incomplete Dates

```
>>> from dateparser import parse
>>> parse('December 2015') # default behavior
datetime.datetime(2015, 12, 16, 0, 0)
>>> parse('December 2015', settings={'PREFER_DAY_OF_MONTH': 'last'})
datetime.datetime(2015, 12, 31, 0, 0)
>>> parse('December 2015', settings={'PREFER_DAY_OF_MONTH': 'first'})
datetime.datetime(2015, 12, 1, 0, 0)
```

```
>>> parse('March')
datetime.datetime(2015, 3, 16, 0, 0)
>>> parse('March', settings={'PREFER_DATES_FROM': 'future'})
datetime.datetime(2016, 3, 16, 0, 0)
>>> # parsing with preference set for 'past'
>>> parse('August', settings={'PREFER_DATES_FROM': 'past'})
datetime.datetime(2015, 8, 15, 0, 0)
```

```
>>> import dateparser
>>> dateparser.parse("2015") # default behavior
datetime.datetime(2015, 3, 27, 0, 0)
>>> dateparser.parse("2015", settings={"PREFER_MONTH_OF_YEAR": "last"})
datetime.datetime(2015, 12, 27, 0, 0)
>>> dateparser.parse("2015", settings={"PREFER_MONTH_OF_YEAR": "first"})
datetime.datetime(2015, 1, 27, 0, 0)
>>> dateparser.parse("2015", settings={"PREFER_MONTH_OF_YEAR": "current"})
datetime.datetime(2015, 3, 27, 0, 0)
```

You can also ignore parsing incomplete dates altogether by setting *STRICT\_PARSING* flag as follows:

```
>>> parse('December 2015', settings={'STRICT_PARSING': True})
None
```

For more on handling incomplete dates, please look at *Settings*.

### Search for Dates in Longer Chunks of Text

#### Warning

Support for searching dates is really limited and needs a lot of improvement, we look forward to community's contribution to get better on that part. See "*Contributing*".

You can extract dates from longer strings of text. They are returned as list of tuples with text chunk containing the date and parsed datetime object.

```
dateparser.search.search_dates(text, languages=None, settings=None, add_detected_language=False,
                                detect_languages_function=None)
```

Find all substrings of the given string which represent date and/or time and parse them.

#### Parameters

- **text** (*str*) – A string in a natural language which may contain date and/or time expressions.
- **languages** (*list*) – A list of two letters language codes.e.g. ['en', 'es']. If languages are given, it will not attempt to detect the language.
- **settings** (*dict*) – Configure customized behavior using settings defined in *dateparser.conf.Settings*.
- **add\_detected\_language** (*bool*) – Indicates if we want the detected language returned in the tuple.
- **detect\_languages\_function** (*function*) – A function for language detection that takes as input a *text* and a *confidence\_threshold*, and returns a list of detected language codes. Note: *detect\_languages\_function* is only uses if *languages* are not provided.

#### Returns

Returns list of tuples containing: substrings representing date and/or time, corresponding *datetime.datetime* object and detected language if *add\_detected\_language* is True. Returns None if no dates that can be parsed are found.

#### Return type

list

**Raises**

ValueError - Unknown Language

```
>>> from dateparser.search import search_dates
>>> search_dates('The first artificial Earth satellite was launched on 4 October.↵
↵1957.')
[('on 4 October 1957', datetime.datetime(1957, 10, 4, 0, 0))]
```

```
>>> search_dates('The first artificial Earth satellite was launched on 4 October.↵
↵1957.',
>>> add_detected_language=True)
[('on 4 October 1957', datetime.datetime(1957, 10, 4, 0, 0), 'en')]
```

```
>>> search_dates("The client arrived to the office for the first time in March.↵
↵3rd, 2004 "
>>> "and got serviced, after a couple of months, on May 6th 2004,↵
↵the customer "
>>> "returned indicating a defect on the part")
[('in March 3rd, 2004 and', datetime.datetime(2004, 3, 3, 0, 0)),
 ('on May 6th 2004', datetime.datetime(2004, 5, 6, 0, 0))]
```

**Advanced Usage**

If you need more control over what is being parser check the *Settings* section as well as the *Using DateDataParser* section.

**Dependencies**

*dateparser* relies on following libraries in some ways:

- *dateutil*'s module *relativedelta* for its freshness parser.
- *convertdate* to convert *Jalali* dates to *Gregorian*.
- *hijridate* to convert *Hijri* dates to *Gregorian*.
- *tzlocal* to reliably get local timezone.
- *ruamel.yaml* (optional) for operations on language files.

**Supported languages and locales**

You can check the supported locales by visiting the “*Supported languages and locales*” section.

**Supported Calendars**

Apart from the Georgian calendar, *dateparser* supports the *Persian Jalali calendar* and the *Hijri/Islami calendar*

To be able to use them you need to install the *calendar* extra by typing:

```
pip install dateparser[calendars]
```

- Example using the *Persian Jalali calendar*. For more information, refer to [Persian Jalali Calendar](#).

```
>>> from dateparser.calendars.jalali import JalaliCalendar
>>> JalaliCalendar('۱۳۸۷ اسفند ۱۳ م سی و چهارم').get_date()
DateData(date_obj=datetime.datetime(2009, 3, 20, 0, 0), period='day', locale=None)
```

- Example using the *Hijri/Islamic Calendar*. For more information, refer to [Hijri Calendar](#).

```
>>> from dateparser.calendars.hijri import HijriCalendar
>>> HijriCalendar('17-01-1437 ٠٥ 08:30 مساءً').get_date()
DateData(date_obj=datetime.datetime(2015, 10, 30, 20, 30), period='day',
↪ locale=None)
```

#### Note

*HijriCalendar* only works with Python  $\geq 3.7$ .

## 2.7.2 Installation

At the command line:

```
$ pip install dateparser
```

Or, if you don't have pip installed:

```
$ easy_install dateparser
```

If you want to install from the latest sources, you can do:

```
$ git clone https://github.com/scrapinghub/dateparser.git
$ cd dateparser
$ python setup.py install
```

## 2.7.3 Using DateDataParser

`dateparser.parse()` uses a default parser which tries to detect language every time it is called and is not the most efficient way while parsing dates from the same source.

`DateDataParser` provides an alternate and efficient way to control language detection behavior.

The instance of `DateDataParser` caches the found languages and will prioritize them when trying to parse the next string.

`dateparser.date.DateDataParser` can also be initialized with known languages:

```
>>> ddp = DateDataParser(languages=['de', 'nl'])
>>> ddp.get_date_data('vr jan 24, 2014 12:49')
DateData(date_obj=datetime.datetime(2014, 1, 24, 12, 49), period='day', locale='nl')
>>> ddp.get_date_data('18.10.14 um 22:56 Uhr')
DateData(date_obj=datetime.datetime(2014, 10, 18, 22, 56), period='day', locale='de')
>>> ddp.get_date_data('11 July 2012')
DateData(date_obj=None, period='day', locale=None)
```

## 2.7.4 Settings

`dateparser`'s parsing behavior can be configured by supplying settings as a dictionary to `settings` argument in `dateparser.parse()` or `DateDataParser` constructor.

#### Note

From *dateparser 1.0.0* when a setting with a wrong value is provided, a `SettingValidationError` is raised.

All supported *settings* with their usage examples are given below:

## Date Order

DATE\_ORDER: specifies the order in which date components *year*, *month* and *day* are expected while parsing ambiguous dates. It defaults to MDY which translates to *month* first, *day* second and *year* last order. Characters *M*, *D* or *Y* can be shuffled to meet required order. For example, DMY specifies *day* first, *month* second and *year* last order:

```
>>> parse('15-12-18 06:00') # assumes default order: MDY
datetime.datetime(2018, 12, 15, 6, 0) # since 15 is not a valid value for Month, it
↳ is swapped with Day's
>>> parse('15-12-18 06:00', settings={'DATE_ORDER': 'YMD'})
datetime.datetime(2015, 12, 18, 6, 0)
```

PREFER\_LOCALE\_DATE\_ORDER: defaults to True. Most languages have a default DATE\_ORDER specified for them. For example, for French it is DMY:

```
>>> # parsing ambiguous date
>>> parse('02-03-2016') # assumes english language, uses MDY date order
datetime.datetime(2016, 2, 3, 0, 0)
>>> parse('1e 02-03-2016') # detects french, hence, uses DMY date order
datetime.datetime(2016, 3, 2, 0, 0)
```

### Note

There's no language level default DATE\_ORDER associated with *en* language. That's why it assumes MDY which is :obj:settings <dateparser.conf.settings> default. If the language has a default DATE\_ORDER associated, supplying custom date order will not be applied unless we set PREFER\_LOCALE\_DATE\_ORDER to False:

```
>>> parse('1e 02-03-2016', settings={'DATE_ORDER': 'MDY'})
datetime.datetime(2016, 3, 2, 0, 0) # MDY didn't apply
```

```
>>> parse('1e 02-03-2016', settings={'DATE_ORDER': 'MDY', 'PREFER_LOCALE_DATE_ORDER
↳ ': False})
datetime.datetime(2016, 2, 3, 0, 0) # MDY worked!
```

## Timezone Related Configurations

TIMEZONE: defaults to local timezone. When specified, resultant datetime is localized with the given timezone. Can be timezone abbreviation or any of [tz database name as given here](#).

```
>>> parse('January 12, 2012 10:00 PM', settings={'TIMEZONE': 'US/Eastern'})
datetime.datetime(2012, 1, 12, 22, 0)
```

TO\_TIMEZONE: defaults to None. When specified, resultant datetime converts according to the supplied timezone:

```
>>> settings = {'TIMEZONE': 'UTC', 'TO_TIMEZONE': 'US/Eastern'}
>>> parse('January 12, 2012 10:00 PM', settings=settings)
datetime.datetime(2012, 1, 12, 17, 0)
```

RETURN\_AS\_TIMEZONE\_AWARE: if True returns tz aware datetime objects in case timezone is detected in the date string.

```
>>> parse('30 mins ago', settings={'RETURN_AS_TIMEZONE_AWARE': True})
datetime.datetime(2017, 3, 13, 1, 43, 10, 243565, tzinfo=<DstTzInfo 'Asia/Karachi' _
↳ PKT+5:00:00 STD>)
```

```
>>> parse('12 Feb 2015 10:56 PM EST', settings={'RETURN_AS_TIMEZONE_AWARE': False})
datetime.datetime(2015, 2, 12, 22, 56)
```

### Handling Incomplete Dates

PREFER\_DAY\_OF\_MONTH: it comes handy when the date string is missing the day part. It defaults to `current` and can be `first` and `last` denoting first and last day of months respectively as values:

```
>>> from dateparser import parse
>>> parse('December 2015') # default behavior
datetime.datetime(2015, 12, 16, 0, 0)
>>> parse('December 2015', settings={'PREFER_DAY_OF_MONTH': 'last'})
datetime.datetime(2015, 12, 31, 0, 0)
>>> parse('December 2015', settings={'PREFER_DAY_OF_MONTH': 'first'})
datetime.datetime(2015, 12, 1, 0, 0)
```

PREFER\_MONTH\_OF\_YEAR: Similarly, another useful thing when the date string is missing the month part. It defaults to `current` and can be `first` and `last` denoting first and last month of year respectively as values:

```
>>> from dateparser import parse
>>> parse("2015") # default behavior
datetime.datetime(2015, 3, 27, 0, 0)
>>> parse("2015", settings={"PREFER_MONTH_OF_YEAR": "last"})
datetime.datetime(2015, 12, 27, 0, 0)
>>> parse("2015", settings={"PREFER_MONTH_OF_YEAR": "first"})
datetime.datetime(2015, 1, 27, 0, 0)
>>> parse("2015", settings={"PREFER_MONTH_OF_YEAR": "current"}) # it exactly behaves
↳ like default one
datetime.datetime(2015, 3, 27, 0, 0)
```

PREFER\_DATES\_FROM: defaults to `current_period` and can have `past` and `future` as values.

If date string is missing some part, this option ensures consistent results depending on the past or future preference, for example, assuming current date is *June 16, 2015*:

```
>>> from dateparser import parse
>>> parse('March')
datetime.datetime(2015, 3, 16, 0, 0)
>>> parse('March', settings={'PREFER_DATES_FROM': 'future'})
datetime.datetime(2016, 3, 16, 0, 0)
>>> # parsing with preference set for 'past'
>>> parse('August', settings={'PREFER_DATES_FROM': 'past'})
datetime.datetime(2015, 8, 15, 0, 0)
```

RELATIVE\_BASE: allows setting the base datetime to use for interpreting partial or relative date strings. Defaults to the current date and time.

For example, assuming current date is *June 16, 2015*:

```
>>> from dateparser import parse
>>> parse('14:30')
datetime.datetime(2015, 6, 16, 14, 30)
>>> parse('14:30', settings={'RELATIVE_BASE': datetime.datetime(2020, 1, 1)})
datetime.datetime(2020, 1, 1, 14, 30)
>>> parse('tomorrow', settings={'RELATIVE_BASE': datetime.datetime(2020, 1, 1)})
datetime.datetime(2020, 1, 2, 0, 0)
```

STRICT\_PARSING: defaults to False.

When set to True if missing any of day, month or year parts, it does not return any result altogether.:

```
>>> parse('March', settings={'STRICT_PARSING': True})
None
```

REQUIRE\_PARTS: ensures results are dates that have all specified parts. It defaults to [] and can include day, month and/or year.

For example, assuming current date is *June 16, 2019*:

```
>>> parse('2012') # default behavior
datetime.datetime(2012, 6, 16, 0, 0)
>>> parse('2012', settings={'REQUIRE_PARTS': ['month']})
None
>>> parse('March 2012', settings={'REQUIRE_PARTS': ['day']})
None
>>> parse('March 12, 2012', settings={'REQUIRE_PARTS': ['day']})
datetime.datetime(2012, 3, 12, 0, 0)
>>> parse('March 12, 2012', settings={'REQUIRE_PARTS': ['day', 'month', 'year']})
datetime.datetime(2012, 3, 12, 0, 0)
```

## Language Detection

SKIP\_TOKENS: it is a list of tokens to discard while detecting language. Defaults to ['t'] which skips T in iso format datetime string .e.g. 2015-05-02T10:20:19+0000.:

```
>>> from dateparser.date import DateDataParser
>>> DateDataParser(settings={'SKIP_TOKENS': ['de']}).get_date_data(u'27 Haziran 1981_↵
↵de') # Turkish (at 27 June 1981)
DateData(date_obj=datetime.datetime(1981, 6, 27, 0, 0), period='day', locale='tr')
```

NORMALIZE: applies unicode normalization (removing accents, diacritics...) when parsing the words. Defaults to True.

```
>>> dateparser.parse('4 decembre 2015', settings={'NORMALIZE': False})
# It doesn't work as the expected input should be '4 décembre 2015'
```

```
>>> dateparser.parse('4 decembre 2015', settings={'NORMALIZE': True})
datetime.datetime(2015, 12, 4, 0, 0)
```

## Default Languages

DEFAULT\_LANGUAGES: It is a list of language codes in ISO 639 that will be used as default languages for parsing when language detection fails. eg. ["en", "fr"]:

```
>>> from dateparser import parse
>>> parse('3 de marzo de 2020', settings={'DEFAULT_LANGUAGES': ["es"]})
```

### Note

When using this setting, these languages will be tried after trying with the detected languages with no success. It is especially useful when using `detect_languages_function`.

## Optional language detection

`LANGUAGE_DETECTION_CONFIDENCE_THRESHOLD`: defaults to 0.5. It is a float of minimum required confidence for the custom language detection:

```
>>> from dateparser import parse
>>> parse('3 de marzo de 2020', settings={'LANGUAGE_DETECTION_CONFIDENCE_THRESHOLD': 0.5}, detect_languages_function=detect_languages)
```

## Other settings

`RETURN_TIME_AS_PERIOD`: returns time as period in date object, if time component is present in date string. Defaults to False.

```
>>> ddp = DateDataParser(settings={'RETURN_TIME_AS_PERIOD': True})
>>> ddp.get_date_data('vr jan 24, 2014 12:49')
DateData(date_obj=datetime.datetime(2014, 1, 24, 12, 49), period='time', locale='nl')
```

`PARSERS`: it is a list of names of parsers to try, allowing to customize which parsers are tried against the input date string, and in which order they are tried.

The following parsers exist:

- `'timestamp'`: If the input string starts with 10 digits, optionally followed by additional digits or a period (`.`), those first 10 digits are interpreted as [Unix time](#).
- `'negative-timestamp'`: `'timestamp'` for negative timestamps. For example, parses `-186454800000` as `1964-02-03T23:00:00`.
- `'relative-time'`: Parses dates and times expressed in relation to the current date and time (e.g. “1 day ago”, “in 2 weeks”).
- `'custom-formats'`: Parses dates that match one of the date formats in the list of the `date_formats` parameter of `dateparser.parse()` or `DateDataParser.get_date_data`.
- `'absolute-time'`: Parses dates and times expressed in absolute form (e.g. “May 4th”, “1991-05-17”). It takes into account settings such as `DATE_ORDER` or `PREFER_LOCALE_DATE_ORDER`.
- `'no-spaces-time'`: Parses dates and times that consist in only digits or a combination of digits and non-digits where the first non-digit it's a colon (e.g. “121994”, “11:052020”). It's not included in the default parsers and it can produce false positives frequently.

`dateparser.settings.default_parsers` contains the default value of `PARSERS` (the list above, in that order) and can be used to write code that changes the parsers to try without skipping parsers that may be added to Dateparser in the future. For example, to ignore relative times:

```
>>> from dateparser_data.settings import default_parsers
>>> parsers = [parser for parser in default_parsers if parser != 'relative-time']
>>> parse('today', settings={'PARSERS': parsers})
```

CACHE\_SIZE\_LIMIT: limits the size of caches, that store data for already processed dates. Default to 1000, but you can set 0 for turning off the limit.

## 2.7.5 Custom language detection

*dateparser* allows to customize the language detection behavior by using the `detect_languages_function` parameter. It currently supports two language detection libraries out of the box: *fastText* and *langdetect*, and allows you to implement your own custom language detection.

### Warning

For short strings the language detection could fail, so it's highly recommended to use `detect_languages_function` along with `DEFAULT_LANGUAGES`.

## Built-in implementations

### fastText

Language detection with fastText.

Import the fastText wrapper and pass it as `detect_languages_function` parameter. Example:

```
>>> from dateparser.custom_language_detection.fasttext import detect_languages
>>> dateparser.parse('12/12/12', detect_languages_function=detect_languages)
```

The fastText integration currently supports the large and the small models. Find more information about *fasttext* models. You can download your model of choice using `dateparser-download`.

Downloading small model:

```
>>> dateparser-download --fasttext small
```

Downloading large model:

```
>>> dateparser-download --fasttext large
```

Deleting all cached models:

```
>>> dateparser-download --clear_cache
```

### Note

If no model has been downloaded, the fastText wrapper downloads and uses the small model by default.

### langdetect

Language detection with langdetect.

Import the langdetect wrapper and pass it as `detect_languages_function` parameter. Example:

```
>>> from dateparser.custom_language_detection.langdetect import detect_languages
>>> dateparser.parse('12/12/12', detect_languages_function=detect_languages)
```

#### Note

From some tests we did, we recommend to use `fastText` for faster and more accurate results.

## Custom implementation

`dateparser` allows the integration of any library to detect languages by wrapping that library in a function that accepts 2 parameters, `text` and `confidence_threshold`, and returns a list of the detected language codes in ISO 639 standards.

Wrapper for boilerplate for implementing custom language detections:

```
def detect_languages(text, confidence_threshold):
    """
    Takes 2 parameters, `text` and `confidence_threshold`, and returns
    a list of `languages codes`.

    * `text` is the input string whose language needs to be detected.

    * `confidence_threshold` is a number between 0 and 1 that indicates the
    minimum confidence required for language matches.

    For language detection libraries that, for each language, indicate how
    confident they are that the language matches the input text, you should
    filter out languages with a confidence lower than this value (adjusted,
    if needed, to the confidence range of the target library).

    This value comes from the dateparser setting
    `LANGUAGE_DETECTION_CONFIDENCE_THRESHOLD`.

    The result must be a list of languages codes (strings).
    """
    # here you can apply your own logic
    return language_codes
```

## 2.7.6 Supported languages and locales

Language	Locales
af	‘af-NA’
agq	
ak	
am	
ar	‘ar-AE’, ‘ar-BH’, ‘ar-DJ’, ‘ar-DZ’, ‘ar-EG’, ‘ar-EH’, ‘ar-ER’, ‘ar-IL’, ‘ar-IQ’, ‘ar-JO’, ‘ar-KM’, ‘ar-KW’, ‘ar-LB’, ‘ar-LY’, ‘ar-M
as	
asa	
ast	
az	
az-Cyrl	

Language	Locales
az-Latn	
bas	
be	
bem	
bez	
bg	
bm	
bn	‘bn-IN’
bo	‘bo-IN’
br	
brx	
bs	
bs-Cyrl	
bs-Latn	
ca	‘ca-AD’, ‘ca-FR’, ‘ca-IT’
ce	
cgg	
chr	
ckb	‘ckb-IR’
cs	
cy	
da	‘da-GL’
dav	
de	‘de-AT’, ‘de-BE’, ‘de-CH’, ‘de-IT’, ‘de-LI’, ‘de-LU’
dje	
dsb	
dua	
dyo	
dz	
ebu	
ee	‘ee-TG’
el	‘el-CY’
en	‘en-001’, ‘en-150’, ‘en-AG’, ‘en-AI’, ‘en-AS’, ‘en-AT’, ‘en-AU’, ‘en-BB’, ‘en-BE’, ‘en-BI’, ‘en-BM’, ‘en-BS’, ‘en-BW’, ‘en-BZ’
eo	
es	‘es-419’, ‘es-AR’, ‘es-BO’, ‘es-BR’, ‘es-BZ’, ‘es-CL’, ‘es-CO’, ‘es-CR’, ‘es-CU’, ‘es-DO’, ‘es-EA’, ‘es-EC’, ‘es-GQ’, ‘es-GT’, ‘es-PE’, ‘es-PR’, ‘es-VA’
et	
eu	
ewo	
fa	‘fa-AF’
ff	‘ff-CM’, ‘ff-GN’, ‘ff-MR’
fi	
fil	
fo	‘fo-DK’
fr	‘fr-BE’, ‘fr-BF’, ‘fr-BI’, ‘fr-BJ’, ‘fr-BL’, ‘fr-CA’, ‘fr-CD’, ‘fr-CF’, ‘fr-CG’, ‘fr-CH’, ‘fr-CI’, ‘fr-CM’, ‘fr-DJ’, ‘fr-DZ’, ‘fr-GA’, ‘fr-GB’, ‘fr-GE’, ‘fr-GH’, ‘fr-GN’, ‘fr-GP’, ‘fr-GQ’, ‘fr-GU’, ‘fr-IV’, ‘fr-JM’, ‘fr-KE’, ‘fr-KM’, ‘fr-LA’, ‘fr-LB’, ‘fr-LC’, ‘fr-LI’, ‘fr-LR’, ‘fr-LU’, ‘fr-LV’, ‘fr-MG’, ‘fr-MI’, ‘fr-ML’, ‘fr-MR’, ‘fr-MU’, ‘fr-MZ’, ‘fr-NC’, ‘fr-NE’, ‘fr-NI’, ‘fr-NL’, ‘fr-NO’, ‘fr-NU’, ‘fr-PE’, ‘fr-PF’, ‘fr-PG’, ‘fr-PH’, ‘fr-PI’, ‘fr-PN’, ‘fr-PT’, ‘fr-RE’, ‘fr-RG’, ‘fr-RH’, ‘fr-RM’, ‘fr-RW’, ‘fr-SB’, ‘fr-SC’, ‘fr-SL’, ‘fr-SM’, ‘fr-SN’, ‘fr-SO’, ‘fr-SR’, ‘fr-ST’, ‘fr-SV’, ‘fr-TD’, ‘fr-TG’, ‘fr-TN’, ‘fr-TL’, ‘fr-TM’, ‘fr-TN’, ‘fr-TS’, ‘fr-TT’, ‘fr-TV’, ‘fr-TZ’, ‘fr-UG’, ‘fr-UN’, ‘fr-UT’, ‘fr-VA’, ‘fr-VE’, ‘fr-VI’, ‘fr-VN’, ‘fr-VU’, ‘fr-WF’, ‘fr-WL’, ‘fr-WT’, ‘fr-WY’, ‘fr-YE’, ‘fr-ZM’, ‘fr-ZW’
fur	
fy	
ga	
gd	
gl	
gsw	‘gsw-FR’, ‘gsw-LI’

Language	Locales
gu	
guz	
gv	
ha	‘ha-GH’, ‘ha-NE’
haw	
he	
hi	
hr	‘hr-BA’
hsb	
hu	
hy	
id	
ig	
ii	
is	
it	‘it-CH’, ‘it-SM’, ‘it-VA’
ja	
jgo	
jmc	
ka	
kab	
kam	
kde	
kea	
khq	
ki	
kk	
kl	
kln	
km	
kn	
ko	‘ko-KP’
kok	
ks	
ksb	
ksf	
ksh	
kw	
ky	
lag	
lb	
lg	
lkt	
ln	‘ln-AO’, ‘ln-CF’, ‘ln-CG’
lo	
lrc	‘lrc-IQ’
lt	
lu	
luo	
luy	

Language	Locales
lv	
mas	‘mas-TZ’
mer	
mfe	
mg	
mgh	
mgo	
mk	
ml	
mn	
mr	
ms	‘ms-BN’, ‘ms-SG’
mt	
mua	
my	
mzn	
naq	
nb	‘nb-SJ’
nd	
ne	‘ne-IN’
nl	‘nl-AW’, ‘nl-BE’, ‘nl-BQ’, ‘nl-CW’, ‘nl-SR’, ‘nl-SX’
nmg	
nn	
nnh	
nus	
nyn	
om	‘om-KE’
or	
os	‘os-RU’
pa	
pa-Arab	
pa-Guru	
pl	
ps	
pt	‘pt-AO’, ‘pt-CH’, ‘pt-CV’, ‘pt-GQ’, ‘pt-GW’, ‘pt-LU’, ‘pt-MO’, ‘pt-MZ’, ‘pt-PT’, ‘pt-ST’, ‘pt-TL’
qu	‘qu-BO’, ‘qu-EC’
rm	
rn	
ro	‘ro-MD’
rof	
ru	‘ru-BY’, ‘ru-KG’, ‘ru-KZ’, ‘ru-MD’, ‘ru-UA’
rw	
rwk	
sah	
saq	
sbp	
se	‘se-FI’, ‘se-SE’
seh	
ses	
sg	

Language	Locales
shi	
shi-Latn	
shi-Tfng	
si	
sk	
sl	
smn	
sn	
so	‘so-DJ’, ‘so-ET’, ‘so-KE’
sq	‘sq-MK’, ‘sq-XK’
sr	
sr-Cyrl	‘sr-Cyrl-BA’, ‘sr-Cyrl-ME’, ‘sr-Cyrl-XK’
sr-Latn	‘sr-Latn-BA’, ‘sr-Latn-ME’, ‘sr-Latn-XK’
sv	‘sv-AX’, ‘sv-FI’
sw	‘sw-CD’, ‘sw-KE’, ‘sw-UG’
ta	‘ta-LK’, ‘ta-MY’, ‘ta-SG’
te	
teo	‘teo-KE’
th	
ti	‘ti-ER’
tl	
to	
tr	‘tr-CY’
twq	
tzm	
ug	
uk	
ur	‘ur-IN’
uz	
uz-Arab	
uz-Cyrl	
uz-Latn	
vi	
vun	
wae	
xog	
yav	
yi	
yo	‘yo-BJ’
yue	
zgh	
zh	
zh-Hans	‘zh-Hans-HK’, ‘zh-Hans-MO’, ‘zh-Hans-SG’
zh-Hant	‘zh-Hant-HK’, ‘zh-Hant-MO’
zu	

## 2.7.7 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

### Types of Contributions

#### Report Bugs

Report bugs at <https://github.com/scrapinghub/dateparser/issues>.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

#### Fix Bugs and Implement Features

Look through the GitHub issues for bugs and feature requests. To avoid duplicate efforts, try to choose issues without related PRs or with staled PRs. We also encourage you to add new languages to the existing stack.

#### Write Documentation

Dateparser could always use more documentation, whether as part of the official Dateparser docs, in docstrings, or even on the web in blog posts, articles, and such.

After you make local changes to the documentation, you will be able to build the project running:

```
tox -e docs
```

Then open `.tox/docs/tmp/html/index.html` in a web browser to see your local build of the documentation.

#### Note

If you don't have `tox` installed, you need to install it first using `pip install tox`.

#### Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/scrapinghub/dateparser/issues>.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that contributions are welcome :)

#### Get Started!

Ready to contribute? Here's how to set up *dateparser* for local development.

1. Fork the *dateparser* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/dateparser.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv dateparser
$ cd dateparser/
$ python setup.py develop
```


4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ tox
```

To get ``tox``, just ``pip install`` it into your virtualenv. In addition to tests,  ``tox`` checks for code style and maximum line length (119 characters).

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

### Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in *README.rst*.
3. Check the pipelines (Github Actions) in the PR comments (or in <https://github.com/scrapinghub/dateparser/actions>) and make sure that the tests pass for all supported Python versions.
4. Check the new project coverage in the PR comments (or in <https://app.codecov.io/gh/scrapinghub/dateparser/pulls>) and make sure that it remained equal or higher than previously.
5. Follow the core developers' advice which aims to ensure code's consistency regardless of the variety of approaches used by many contributors.
6. In case you are unable to continue working on a PR, please leave a short comment to notify us. We will be pleased to make any changes required to get it done.

### Guidelines for Editing Translation Data

English is the primary language of Dateparser. Dates in all other languages are translated into English equivalents before they are parsed.

The language data that Dateparser uses to parse dates is in `dateparser/data/date_translation_data`. For each supported language, there is a Python file containing translation data.

Each translation data Python files contains different kinds of translation data for date parsing: month and week names - and their abbreviations, prepositions, conjunctions, frequently used descriptive words and phrases (like “today”), etc.

Translation data Python files are generated from the following sources:

- **Unicode CLDR** data in JSON format, located at `dateparser_data/cldr_language_data/date_translation_data`
- Additional data from the Dateparser community in YAML format, located at `dateparser_data/supplementary_language_data/date_translation_data`

If you wish to extend the data of an existing language, or add data for a new language, you must:

1. Edit or create the corresponding file within `dateparser_data/supplementary_language_data/date_translation_data`

See existing files to learn how they are defined, and see [Language Data Template](#) for details.

2. Regenerate the corresponding file within `dateparser/data/date_translation_data` running the following script:

```
dateparser_scripts/write_complete_data.py
```

3. Write tests that cover your changes

You should be able to find tests that cover the affected data, and use copy-and-paste to create the corresponding new test.

If in doubt, ask Dateparser maintainers for help.

## Language Data Template

```
two-letter language code as defined in ISO-639-1 (https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes). e.g. for English - en:
name: language name (e.g. English)
no_word_spacing: False (set to True for languages that do not use spaces between words)

skip: ["words", "to", "skip", "such", "as", "and", "or", "at", "in", "alphabetical", "order"]

pertain: []

monday:
  - name for Monday
  - abbreviation for Monday
tuesday:
  - as above
wednesday:
  - as above
thursday:
  - as above
friday:
  - as above
saturday:
  - as above
sunday:
  - as above
```

(continues on next page)

(continued from previous page)

```
january:
  - name for January
  - abbreviation for January
february:
  - as above
march:
  - as above
april:
  - as above
may:
  - as above
june:
  - as above
july:
  - as above
august:
  - as above
september:
  - as above
october:
  - as above
november:
  - as above
december:
  - as above

year:
  - name for year
  - abbreviation for year
month:
  - as above
week:
  - as above
day:
  - as above
hour:
  - as above
minute:
  - as above
second:
  - as above

ago:
  - words that stand
  - for "ago"

simplifications:
  - word: replacement
  - regex: replacement
  - day before yesterday: 2 days ago
```

## Updating the List of Supported Languages and Locales

Whenever the content of `dateparser.data.languages_info.language_locale_dict` is modified, use `dateparser_scripts/update_supported_languages_and_locales.py` to update the corresponding documentation table:

```
dateparser_scripts/update_supported_languages_and_locales.py
```

## 2.7.8 API reference

### dateparser package

#### Subpackages

#### dateparser.languages package

#### Submodules

#### dateparser.languages.dictionary module

**class** `dateparser.languages.dictionary.Dictionary(locale_info, settings=None)`

Bases: `object`

Class that modifies and stores translations and handles splitting of date string.

#### Parameters

- **locale\_info** – Locale info (translation data) of the locale.
- **settings** (*dict*) – Configure customized behavior using settings defined in `dateparser.conf.Settings`.

#### Returns

a Dictionary instance.

**are\_tokens\_valid** (*tokens*)

Check if tokens are valid tokens for the locale.

#### Parameters

**tokens** (*list*) – a list of string tokens.

#### Returns

True if tokens are valid, False otherwise.

**split** (*string*, *keep\_formatting=False*)

Split the date string using translations in locale info.

#### Parameters

- **string** (*str*) – Date string to be splitted.
- **keep\_formatting** (*bool*) – If True, retain formatting of the date string.

#### Returns

A list of string tokens formed after splitting the date string.

**class** `dateparser.languages.dictionary.NormalizedDictionary(locale_info, settings=None)`

Bases: `Dictionary`

**exception** `dateparser.languages.dictionary.UnknownTokenError`

Bases: `Exception`

## dateparser.languages.loader module

**class** dateparser.languages.loader.LocaleDataLoader

Bases: object

Class that handles loading of locale instances.

**get\_locale** (*shortname*)

Get a locale instance.

### Parameters

**shortname** (*str*) – A locale code, e.g. ‘fr-PF’, ‘qu-EC’, ‘af-NA’.

### Returns

locale instance

**get\_locale\_map** (*languages=None, locales=None, region=None, use\_given\_order=False, allow\_conflicting\_locales=False*)

Get an ordered mapping with locale codes as keys and corresponding locale instances as values.

### Parameters

- **languages** (*list*) – A list of language codes, e.g. [‘en’, ‘es’, ‘zh-Hant’]. If locales are not given, languages and region are used to construct locales to load.
- **locales** (*list*) – A list of codes of locales which are to be loaded, e.g. [‘fr-PF’, ‘qu-EC’, ‘af-NA’]
- **region** (*str*) – A region code, e.g. ‘IN’, ‘001’, ‘NE’. If locales are not given, languages and region are used to construct locales to load.
- **use\_given\_order** (*bool*) – If True, the returned mapping is ordered in the order locales are given.
- **allow\_conflicting\_locales** (*bool*) – if True, locales with same language and different region can be loaded.

### Returns

ordered locale code to locale instance mapping

**get\_locales** (*languages=None, locales=None, region=None, use\_given\_order=False, allow\_conflicting\_locales=False*)

Yield locale instances.

### Parameters

- **languages** (*list*) – A list of language codes, e.g. [‘en’, ‘es’, ‘zh-Hant’]. If locales are not given, languages and region are used to construct locales to load.
- **locales** (*list*) – A list of codes of locales which are to be loaded, e.g. [‘fr-PF’, ‘qu-EC’, ‘af-NA’]
- **region** (*str*) – A region code, e.g. ‘IN’, ‘001’, ‘NE’. If locales are not given, languages and region are used to construct locales to load.
- **use\_given\_order** (*bool*) – If True, the returned mapping is ordered in the order locales are given.
- **allow\_conflicting\_locales** (*bool*) – if True, locales with same language and different region can be loaded.

### Yield

locale instances

## dateparser.languages.locale module

**class** dateparser.languages.locale.Locale (*shortname, language\_info*)

Bases: object

Class that deals with applicability and translation from a locale.

### Parameters

- **shortname** (*str*) – A locale code, e.g. 'fr-PF', 'qu-EC', 'af-NA'.
- **language\_info** (*dict*) – Language info (translation data) of the language the locale belongs to.

### Returns

A Locale instance

**static clean\_dictionary** (*dictionary, threshold=2*)

**count\_applicability** (*text, strip\_timezone=False, settings=None*)

**get\_wordchars\_for\_detection** (*settings*)

**is\_applicable** (*date\_string, strip\_timezone=False, settings=None*)

Check if the locale is applicable to translate date string.

### Parameters

- **date\_string** (*str*) – A string representing date and/or time in a recognizably valid format.
- **strip\_timezone** (*bool*) – If True, timezone is stripped from date string.

### Returns

boolean value representing if the locale is applicable for the date string or not.

**to\_parserinfo** (*base\_cls=<class 'dateutil.parser.\_parser.parserinfo'>*)

**translate** (*date\_string, keep\_formatting=False, settings=None*)

Translate the date string to its English equivalent.

### Parameters

- **date\_string** (*str*) – A string representing date and/or time in a recognizably valid format.
- **keep\_formatting** (*bool*) – If True, retain formatting of the date string after translation.

### Returns

translated date string.

**translate\_search** (*search\_string, settings=None*)

## dateparser.languages.validation module

**class** dateparser.languages.validation.LanguageValidator

Bases: object

```
VALID_KEYS = ['name', 'skip', 'pertain', 'simplifications', 'no_word_spacing',
'ago', 'in', 'monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'saturday',
'sunday', 'january', 'february', 'march', 'april', 'may', 'june', 'july',
'august', 'september', 'october', 'november', 'december', 'year', 'month', 'week',
'day', 'hour', 'minute', 'second', 'sentence_splitter_group']
```

```
classmethod get_logger()  
  
logger = None  
  
classmethod validate_info(language_id, info)
```

### Module contents

#### dateparser.calendars package

#### Submodules

#### Module contents

```
class dateparser.calendars.CalendarBase(source)  
    Bases: object  
  
    Base setup class for non-Gregorian calendar system.  
  
    Parameters  
        source (str) – Date string passed to calendar parser.
```

#### Submodules

#### dateparser.conf module

```
exception dateparser.conf.SettingValidationError  
    Bases: ValueError  
  
class dateparser.conf.Settings(*args, **kwargs)  
    Bases: object  
  
    Control and configure default parsing behavior of dateparser. Currently, supported settings are:  
  
    • DATE_ORDER  
    • PREFER_LOCALE_DATE_ORDER  
    • TIMEZONE  
    • TO_TIMEZONE  
    • RETURN_AS_TIMEZONE_AWARE  
    • PREFER_MONTH_OF_YEAR  
    • PREFER_DAY_OF_MONTH  
    • PREFER_DATES_FROM  
    • RELATIVE_BASE  
    • STRICT_PARSING  
    • REQUIRE_PARTS  
    • SKIP_TOKENS  
    • NORMALIZE  
    • RETURN_TIME_AS_PERIOD  
    • PARSERS
```

- `DEFAULT_LANGUAGES`
- `LANGUAGE_DETECTION_CONFIDENCE_THRESHOLD`
- `CACHE_SIZE_LIMIT`

`classmethod get_key (settings=None)`

`replace (mod_settings=None, **kws)`

`dateparser.conf.apply_settings (f)`

`dateparser.conf.check_settings (settings)`

Check if provided settings are valid, if not it raises *SettingValidationError*. Only checks for the modified settings.

## dateparser.date module

`class dateparser.date.DateData (* (Keyword-only parameters separator (PEP 3102)), date_obj=None, period=None, locale=None)`

Bases: `object`

Class that represents the parsed data with useful information. It can be accessed with square brackets like a dict object.

`class dateparser.date.DateDataParser (languages=None, locales=None, region=None, try_previous_locales=False, use_given_order=False, settings=None, detect_languages_function=None)`

Bases: `object`

Class which handles language detection, translation and subsequent generic parsing of string representing date and/or time.

### Parameters

- **languages** (*list*) – A list of language codes, e.g. ['en', 'es', 'zh-Hant']. If locales are not given, languages and region are used to construct locales for translation.
- **locales** (*list*) – A list of locale codes, e.g. ['fr-PF', 'qu-EC', 'af-NA']. The parser uses only these locales to translate date string.
- **region** (*str*) – A region code, e.g. 'IN', '001', 'NE'. If locales are not given, languages and region are used to construct locales for translation.
- **try\_previous\_locales** (*bool*) – If True, locales previously used to translate date are tried first.
- **use\_given\_order** (*bool*) – If True, locales are tried for translation of date string in the order in which they are given.
- **settings** (*dict*) – Configure customized behavior using settings defined in `dateparser.conf.Settings`.
- **detect\_languages\_function** (*function*) – A function for language detection that takes as input a *text* and a *confidence\_threshold*, and returns a list of detected language codes. Note: this function is only used if `languages` and `locales` are not provided.

### Returns

A parser instance

### Raises

`ValueError`: Unknown Language, `TypeError`: Languages argument must be a list, `SettingValidationError`: A provided setting is not valid.

**get\_date\_data** (*date\_string*, *date\_formats=None*)

Parse string representing date and/or time in recognizable localized formats. Supports parsing multiple languages and timezones.

#### Parameters

- **date\_string** (*str*) – A string representing date and/or time in a recognizably valid format.
- **date\_formats** (*list*) – A list of format strings using directives as given [here](#). The parser applies formats one by one, taking into account the detected languages.

#### Returns

a `DateData` object.

#### Raises

`ValueError` - Unknown Language

#### Note

*Period* values can be a 'day' (default), 'week', 'month', 'year', 'time'.

*Period* represents the granularity of date parsed from the given string.

In the example below, since no day information is present, the day is assumed to be current day 16 from *current date* (which is June 16, 2015, at the moment of writing this). Hence, the level of precision is month:

```
>>> DateDataParser().get_date_data('March 2015')
DateData(date_obj=datetime.datetime(2015, 3, 16, 0, 0), period='month',
↪ locale='en')
```

Similarly, for date strings with no day and month information present, level of precision is *year* and day 16 and month 6 are from *current date*.

```
>>> DateDataParser().get_date_data('2014')
DateData(date_obj=datetime.datetime(2014, 6, 16, 0, 0), period='year', locale=
↪ 'en')
```

Dates with time zone indications or UTC offsets are returned in UTC time unless specified using [Settings](#).

```
>>> DateDataParser().get_date_data('23 March 2000, 1:21 PM CET')
DateData(date_obj=datetime.datetime(2000, 3, 23, 13, 21, tzinfo=<StaticTzInfo
↪ 'CET'>),
period='day', locale='en')
```

**get\_date\_tuple** (*\*args*, *\*\*kwargs*)

**locale\_loader** = None

`dateparser.date.date_range` (*begin*, *end*, *\*\*kwargs*)

`dateparser.date.get_date_from_timestamp` (*date\_string*, *settings*, *negative=False*)

`dateparser.date.get_intersecting_periods` (*low*, *high*, *period='day'*)

`dateparser.date.parse_with_formats` (*date\_string*, *date\_formats*, *settings*)

Parse with formats and return a dictionary with 'period' and 'obj\_date'.

**Returns**

`datetime.datetime`, `dict` or `None`

`dateparser.date.sanitize_date` (*date\_string*)

`dateparser.date.sanitize_spaces` (*date\_string*)

**dateparser.date\_parser module**

**class** `dateparser.date_parser.DateParser`

Bases: `object`

**parse** (*date\_string*, *parse\_method*, *settings=None*)

**dateparser.freshness\_date\_parser module**

**class** `dateparser.freshness_date_parser.FreshnessDateDataParser`

Bases: `object`

Parses date string like “1 year, 2 months ago” and “3 hours, 50 minutes ago”

**get\_date\_data** (*date\_string*, *settings=None*)

**get\_kwargs** (*date\_string*)

**get\_local\_tz** ()

**parse** (*date\_string*, *settings*)

**dateparser.timezone\_parser module**

**class** `dateparser.timezone_parser.StaticTzInfo` (*name*, *offset*)

Bases: `tzinfo`

**dst** (*dt*)

`datetime` -> DST offset as `timedelta` positive east of UTC.

**localize** (*dt*, *is\_dst=False*)

**tzname** (*dt*)

`datetime` -> string name of time zone.

**utcoffset** (*dt*)

`datetime` -> `timedelta` showing offset from UTC, negative values indicating West of UTC

`dateparser.timezone_parser.build_tz_offsets` (*search\_regex\_parts*)

`dateparser.timezone_parser.convert_to_local_tz` (*datetime\_obj*, *datetime\_tz\_offset*)

`dateparser.timezone_parser.get_local_tz_offset` ()

`dateparser.timezone_parser.pop_tz_offset_from_string` (*date\_string*, *as\_offset=True*)

`dateparser.timezone_parser.word_is_tz` (*word*)

## dateparser.timezones module

## dateparser.utils module

`dateparser.utils.apply_dateparser_timezone(utc_datetime, offset_or_timezone_abb)`

`dateparser.utils.apply_timezone(date_time, tz_string)`

`dateparser.utils.apply_timezone_from_settings(date_obj, settings)`

`dateparser.utils.apply_tzdatabase_timezone(date_time, pytz_string)`

`dateparser.utils.combine_dicts(primary_dict, supplementary_dict)`

`dateparser.utils.find_date_separator(format)`

`dateparser.utils.get_last_day_of_month(year, month)`

`dateparser.utils.get_logger()`

`dateparser.utils.get_next_leap_year(year)`

`dateparser.utils.get_previous_leap_year(year)`

`dateparser.utils.get_timezone_from_tz_string(tz_string)`

`dateparser.utils.localize_timezone(date_time, tz_string)`

`dateparser.utils.normalize_unicode(string, form='NFKD')`

`dateparser.utils.registry(cls)`

`dateparser.utils.set_correct_day_from_settings(date_obj, settings, current_day=None)`  
Set correct day attending the *PREFER\_DAY\_OF\_MONTH* setting.

`dateparser.utils.set_correct_month_from_settings(date_obj, settings, current_month=None)`  
Set correct month attending the *PREFER\_MONTH\_OF\_YEAR* setting.

`dateparser.utils.setup_logging()`

`dateparser.utils.strip_braces(date_string)`

## Module contents

`dateparser.parse(date_string, date_formats=None, languages=None, locales=None, region=None, settings=None, detect_languages_function=None)`

Parse date and time from given date string.

### Parameters

- **date\_string** (*str*) – A string representing date and/or time in a recognizably valid format.
- **date\_formats** (*list*) – A list of format strings using directives as given [here](#). The parser applies formats one by one, taking into account the detected languages/locales.
- **languages** (*list*) – A list of language codes, e.g. ['en', 'es', 'zh-Hant']. If locales are not given, languages and region are used to construct locales for translation.
- **locales** (*list*) – A list of locale codes, e.g. ['fr-PF', 'qu-EC', 'af-NA']. The parser uses only these locales to translate date string.

- **region** (*str*) – A region code, e.g. ‘IN’, ‘001’, ‘NE’. If locales are not given, languages and region are used to construct locales for translation.
- **settings** (*dict*) – Configure customized behavior using settings defined in `dateparser.conf.Settings`.
- **detect\_languages\_function** (*function*) – A function for language detection that takes as input a string (the *date\_string*) and a *confidence\_threshold*, and returns a list of detected language codes. Note: this function is only used if `languages` and `locales` are not provided.

**Returns**

Returns `datetime` representing parsed date if successful, else returns `None`

**Return type**

`datetime`.

**Raises**

`ValueError: Unknown Language`, `TypeError: Languages argument must be a list`, `Setting-ValidationError: A provided setting is not valid`.

## 2.7.9 Credits

Currently, more than 100 committers have contributed to this project, making this contributors list really hard to maintain, so we have decided to stop updating this list.

To see the people behind this code, you can run `git shortlog -s -n` or visit the contributions section in Github: <https://github.com/scrapinghub/dateparser/graphs/contributors>

We really appreciate **all the people that has contributed to this project with their time and ideas**. Special mention to **Waqas Shabir** (waqasshabbir), **Eugene Amirov** (Allactaga) and **Artur Sadurski** (asadurski) for creating and maintaining this awesome project.

To all of you... thank you for building and improving this!

## 2.7.10 History

### 1.2.1 (2025-02-05)

Fixes:

- Fix `PytzUsageWarning` (#1109)
- Fix `date_parser` with `prefer_month_of_year` wrong results (#1224)
- Fix skipped day when UTC and tz are different days (#1183)

Improvements:

- Avoid repeated loop over timezones (#1238)
- Proofread `README.rst` (#1234)
- Check for derived types for configuration (#1223)
- Parse some abbreviated strings as relative dates (#1219)
- Migrate from `hijri-converter` to `hijridate` (#1211)
- Fixed `ClusterFuzz` build error by adding `dateparser.data` as a binary (#1208)
- Fix an issue detected by `OSSFuzz` (#1203)
- Support two-digit years in non-Gregorian calendars (#1187)

- Refactored CI to run extras separately and test minimum versions of dependencies, replaced flake8 with ruff, fixed tests (#1248)
- Set minimum versions for dependencies (#1248)
- Limited `numpy` to 1.x when installing `dateparser[fasttext]` (#1248)

### 1.2.0 (2023-11-17)

New features:

- New `PREFER_MONTH_OF_YEAR` setting (#1146)

Fixes:

- Absolute years in Russian are no longer being treated as a number of years in the past (#1129)

Cleanups and internal improvements:

- Removed the use of `datetime.utcnow`, deprecated on Python 3.12 (#1179)
- Applied Black formatting to the code base (#1158)
- Initial integration with OSSFuzz (#1198)
- Extended test cases (#1191)

### 1.1.8 (2023-03-22)

Improvements:

- Improved date parsing for Chinese (#1148)
- Improved date parsing for Czech (#1151)
- Reorder language by popularity (#1152)
- Fix leak of memory in cache (#1140)
- Add support for “d units later” (#1154)
- Move modification in CLDR data to yaml (#1153)
- Add support to use timezone via settings to get `PREFER_DATES_FROM` result (#1155)

### 1.1.7 (2023-02-02)

Improvements:

- Add an “ago” synonym for Arabic (#1128)
- Improved date parsing for Czech (#1131)
- Improved date parsing for Indonesian (#1134)

### 1.1.6 (2023-01-12)

Improvements:

- Fix the bug where Monday is parsed as a month (#1121)
- Prevent ReDoS in Spanish sentence splitting regex (#1084)

### 1.1.5 (2022-12-29)

Improvements:

- Parse short versions of day, month, and year (#1103)
- Add a test for “in 1d” (#1104)
- Update languages\_info (#1107)
- Add a workaround for zipimporter not having exec\_module before Python 3.10 (#1069)
- Stabilize tests at midnight (#1111)
- Add a test case for French (#1110)

Cleanups:

- Remove the requirements-build file (#1113)

### 1.1.4 (2022-11-21)

Improvements:

- Improved support for languages such as Slovak, Indonesian, Hindi, German and Japanese (#1064, #1094, #986, #1071, #1068)
- Recursively create a model home (#996)
- Replace regex sub with simple string replace (#1095)
- Add Python 3.10, 3.11 support (#1096)
- Drop support for Python 3.5, 3.6 versions (#1097)

### 1.1.3 (2022-11-03)

New features:

- Add support for fractional units (#876)

Improvements:

- Fix the returned datetime skipping a day with time+timezone input and PREFER\_DATES\_FROM = ‘future’ (#1002)
- Fix input translation breaking keep\_formatting (#720)
- English: support “till date” (#1005)
- English: support “after” and “before” in relative dates (#1008)

Cleanups:

- Reorganize internal data (#1090)
- CI updates (#1088)

### 1.1.2 (2022-10-20)

Improvements:

- Added support for negative timestamp (#1060)
- Fixed PytzUsageWarning for Python versions >= 3.6 (#1062)
- Added support for dates with dots and spaces (#1028)

- Improved support for Ukrainian, Croatian and Russian (#1072, #1074, #1079, #1082, #1073, #1083)
- Added support for parsing Unix timestamps consistently regardless of timezones (#954)
- Improved tests (#1086)

### 1.1.1 (2022-03-17)

Improvements:

- Fixed issue with regex library by pinning dependencies to an earlier version (< 2022.3.15, #1046).
- Extended support for Russian language dates starting with lowercase (#999).
- Allowed to use `_given_order` for languages too (#997).
- Fixed link to settings section (#1018).
- Defined UTF-8 encoding for Windows (#998).
- Fixed directories creation error in CLI utils (#1022).

### 1.1.0 (2021-10-04)

New features:

- Support language detection based on `langdetect`, `fastText`, or a custom implementation (see #932)
- Add support for 'by <time>' (see #839)
- Sort default language list by internet usage (see #805)

Improvements:

- Improved support of Chinese (#910), Czech (#977)
- Improvements in `search_dates` (see #953)
- Make order of previous locales deterministic (see #851)
- Fix parsing with trailing space (see #841)
- Consider `RETURN_TIME_AS_PERIOD` for timestamp times (see #922)
- Exclude failing regex version (see #974)
- Ongoing work multithreading support (see #881, #885)
- Add demo URL (see #883)

QA:

- Migrate pipelines from Travis CI to Github Actions (see #859, #879, #884, #886, #911, #966)
- Use versioned CLDR data (see #825)
- Add a script to update table of supported languages and locales (see #601)
- Sort 'skip' keys in yaml files (see #844)
- Improve test coverage (see #827)
- Code cleanup (see #888, #907, #951, #958, #957)

## 1.0.0 (2020-10-29)

### Breaking changes:

- Drop support for Python 2.7 and pypy (see #727, #744, #748, #749, #754, #755, #758, #761, #763, #764, #777 and #783)
- Now `DateDataParser.get_date_data()` returns a `DateData` object instead of a dict (see #778).
- From now wrong settings are not silenced and raise `SettingValidationError` (see #797)
- Now `dateparser.parse()` is deterministic and doesn't try previous locales. Also, `DateDataParser.get_date_data()` doesn't try the previous locales by default (see #781)
- Remove the 'base-formats' parser (see #721)
- Extract the 'no-spaces-time' parser from the 'absolute-time' parser and make it an optional parser (see #786)
- Remove `numeral_translation_data` (see #782)
- Remove the undocumented `SKIP_TOKENS_PARSER` and `FUZZY` settings (see #728, #794)
- Remove support for using strings in `date_formats` (see #726)
- The undocumented `ExactLanguageSearch` class has been moved to the private scope and some internal methods have changed (see #778)
- Changes in `dateparser.utils`: `normalize_unicode()` doesn't accept bytes as input and `convert_to_unicode` has been deprecated (see #749)

### New features:

- Add Python 3.9 support (see #732, #823)
- Detect hours separated with a period/dot (see #741)
- Add support for “decade” (see #762)
- Add support for the hijri calendar in Python  $\geq 3.6$  (see #718)

### Improvements:

- New logo! (see #719)
- Improve the README and docs (see #779, #722)
- Fix the “calendars” extra (see #740)
- Fix leap years when `PREFER_DATES_FROM` is set (see #738)
- Fix `STRICT_PARSING` setting in `no-spaces-time` parser (see #715)
- Consider `RETURN_AS_TIME_PERIOD` setting for `relative-time` parser (see #807)
- Parse the 24hr time format with meridian info (see #634)
- Other small improvements (see #698, #709, #710, #712, #730, #731, #735, #739, #784, #788, #795 and #801)

## 0.7.6 (2020-06-12)

### Improvements:

- Rename `scripts` to `dateparser_scripts` to avoid name collisions with modules from other packages or projects (see #707)

### 0.7.5 (2020-06-10)

New features:

- Add Python 3.8 support (see #664)
- Implement a `REQUIRE_PARTS` setting (see #703)
- Add support for subscript and superscript numbers (see #684)
- Extended French support (see #672)
- Extended German support (see #673)

Improvements:

- Migrate test suite to Pytest (see #662)
- Add test to check the *yaml* and *json* files content (see #663 and #692)
- Add flake8 pipeline with pytest-flake8 (see #665)
- Add partial support for 8-digit dates without separators (see #639)
- Fix possible `OverflowError` errors and explicitly avoid to raise `ValueError` when parsing relative dates (see #686)
- Fix double-digit GMT and UTC parsing (see #632)
- Fix bug when using `DATE_ORDER` (see #628)
- Fix bug when parsing relative time with timezone (see #503)
- Fix milliseconds parsing (see #572 and #661)
- Fix wrong values to be interpreted as 'future' in `PREFER_DATES_FROM` (see #629)
- Other small improvements (see #667, #675, #511, #626, #512, #509, #696, #702 and #699)

### 0.7.4 (2020-03-06)

New features:

- Extended Norwegian support (see #598)
- Implement a `PARSERS` setting (see #603)

Improvements:

- Add support for `PREFER_DATES_FROM` in relative/freshness parser (see #414)
- Add support for `PREFER_DAY_OF_MONTH` in base-formats parser (see #611)
- Added UTC -00:00 as a valid offset (see #574)
- Fix support for “one” (see #593)
- Fix `TypeError` when parsing some invalid dates (see #536)
- Fix tokenizer for non recognized characters (see #622)
- Prevent installing regex 2019.02.19 (see #600)
- Resolve `DeprecationWarning` related to raw string escape sequences (see #596)
- Implement a tox environment to build the documentation (see #604)
- Improve tests stability (see #591, #605)
- Documentation improvements (see #510, #578, #619, #614, #620)

- Performance improvements (see #570, #569, #625)

### 0.7.3 (2020-03-06)

- Broken version

### 0.7.2 (2019-09-17)

Features:

- Extended Czech support
- Added `time` to valid periods
- Added timezone information to dates found with `search_dates()`
- Support strings as date formats

Improvements:

- Fixed Collections ABCs depreciation warning
- Fixed dates with trailing colons not being parsed
- Fixed date format override on any settings change
- Fixed parsing current weekday as past date, regardless of settings
- Added UTC -2:30 as a valid offset
- Added Python 3.7 to supported versions, dropped support for Python 3.3 and 3.4
- Moved to `importlib` from `imp` where possible
- Improved support for Catalan
- Documentation improvements

### 0.7.1 (2019-02-12)

Features/news:

- Added detected language to return value of `search_dates()`
- Performance improvements
- Refreshed versions of dependencies

Improvements:

- Fixed unpickleable `DateTime` objects with timezones
- Fixed regex pattern to avoid new behaviour of `re.split` in Python 3.7
- Fixed an exception thrown when parsing colons
- Fixed tests failing on days with number greater than 30
- Fixed `ZeroDivisionError` exceptions

### 0.7.0 (2018-02-08)

Features added during Google Summer of Code 2017:

- Harvesting language data from Unicode CLDR database (<https://github.com/unicode-cldr/cldr-json>), which includes over 200 locales (#321) - authored by Sarthak Maddan. See full currently supported locale list in README.

- Extracting dates from longer strings of text (#324) - authored by Elena Zakharova. Special thanks for their awesome contributions!

New features:

- Added (independently from CLDR) Georgian (#308) and Swedish (#305)

Improvements:

- Improved support of Chinese (#359), Thai (#345), French (#301, #304), Russian (#302)
- Removed ruamel.yaml from dependencies (#374). This should reduce the number of installation issues and improve performance as the result of moving away from YAML as basic data storage format. Note that YAML is still used as format for support language files.
- Improved performance through using pre-compiling frequent regexes and lazy loading of data (#293, #294, #295, #315)
- Extended tests (#316, #317, #318, #323)
- Updated nose\_parameterized to its current package, parameterized (#381)

Planned for next release:

- Full language and locale names
- Performance and stability improvements
- Documentation improvements

### 0.6.0 (2017-03-13)

New features:

- Consistent parsing in terms of true python representation of date string. See #281
- Added support for Bangla, Bulgarian and Hindi languages.

Improvements:

- Major bug fixes related to parser and system's locale. See #277, #282
- Type check for timezone arguments in settings. see #267
- Pinned dependencies' versions in requirements. See #265
- Improved support for cn, es, dutch languages. See #274, #272, #285

Packaging:

- Make calendars extras to be used at the time of installation if need to use calendars feature.

### 0.5.1 (2016-12-18)

New features:

- Added support for Hebrew

Improvements:

- Safer loading of YAML. See #251
- Better timezone parsing for freshness dates. See #256
- Pinned dependencies' versions in requirements. See #265
- Improved support for zh, fi languages. See #249, #250, #248, #244

### 0.5.0 (2016-09-26)

New features:

- `DateDataParser` now also returns detected language in the result dictionary.
- Explicit and lucid timezone conversion for a given datestring using `TIMEZONE`, `TO_TIMEZONE` settings.
- Added Hungarian language.
- Added setting, `STRICT_PARSING` to ignore incomplete dates.

Improvements:

- Fixed quite a few parser bugs reported in issues #219, #222, #207, #224.
- Improved support for chinese language.
- Consistent interface for both Jalali and Hijri parsers.

### 0.4.0 (2016-06-17)

New features:

- Support for Language based date order preference while parsing ambiguous dates.
- Support for parsing dates with no spaces in between components.
- Support for custom date order preference using `settings`.
- Support for parsing generic relative dates in future.e.g. “tomorrow”, “in two weeks”, etc.
- Added `RELATIVE_BASE` settings to set date context to any datetime in past or future.
- Replaced `dateutil.parser.parse` with `dateparser`’s own parser.

Improvements:

- Added simplifications for “12 noon” and “12 midnight”.
- Fixed several bugs
- Replaced PyYAML library by its active fork *ruamel.yaml* which also fixed the issues with installation on windows using python35.
- More predictable `date_formats` handling.

### 0.3.5 (2016-04-27)

New features:

- Danish language support.
- Japanese language support.
- Support for parsing date strings with accents.

Improvements:

- Transformed `languages.yaml` into base file and separate files for each language.
- Fixed vietnamese language simplifications.
- No more version restrictions for `python-dateutil`.
- Timezone parsing improvements.
- Fixed test environments.

- Cleaned language codes. Now we strictly follow codes as in ISO 639-1.
- Improved chinese dates parsing.

### 0.3.4 (2016-03-03)

Improvements:

- Fixed broken version 0.3.3 by excluding latest python-dateutil version.

### 0.3.3 (2016-02-29)

New features:

- Finnish language support.

Improvements:

- Faster parsing with switching to regex module.
- `RETURN_AS_TIMEZONE_AWARE` setting to return tz aware date object.
- Fixed conflicts with month/weekday names similarity across languages.

### 0.3.2 (2016-01-25)

New features:

- Added Hijri Calendar support.
- Added settings for better control over parsing dates.
- Support to convert parsed time to the given timezone for both complete and relative dates.

Improvements:

- Fixed problem with caching `datetime.now()` in `FreshnessDateDataParser`.
- Added month names and week day names abbreviations to several languages.
- More simplifications for Russian and Ukrainian languages.
- Fixed problem with parsing time component of date strings with several kinds of apostrophes.

### 0.3.1 (2015-10-28)

New features:

- Support for Jalali Calendar.
- Belarusian language support.
- Indonesian language support.

Improvements:

- Extended support for Russian and Polish.
- Fixed bug with time zone recognition.
- Fixed bug with incorrect translation of “second” for Portuguese.

### 0.3.0 (2015-07-29)

New features:

- Compatibility with Python 3 and PyPy.

Improvements:

- *languages.yaml* data cleaned up to make it human-readable.
- Improved Spanish date parsing.

### 0.2.1 (2015-07-13)

- Support for generic parsing of dates with UTC offset.
- Support for Tagalog/Filipino dates.
- Improved support for French and Spanish dates.

### 0.2.0 (2015-06-17)

- Easy to use `parse` function
- Languages definitions using YAML.
- Using translation based approach for parsing non-english languages. Previously, `dateutil.parserinfo` was used for language definitions.
- Better period extraction.
- Improved tests.
- Added a number of new simplifications for more comprehensive generic parsing.
- Improved validation for dates.
- Support for Polish, Thai and Arabic dates.
- Support for `pytz` timezones.
- Fixed building and packaging issues.

### 0.1.0 (2014-11-24)

- First release on PyPI.
- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### d

- `dateparser`, [40](#)
- `dateparser.calendars`, [36](#)
- `dateparser.conf`, [36](#)
- `dateparser.date`, [37](#)
- `dateparser.date_parser`, [39](#)
- `dateparser.freshness_date_parser`, [39](#)
- `dateparser.languages`, [36](#)
- `dateparser.languages.dictionary`, [33](#)
- `dateparser.languages.loader`, [34](#)
- `dateparser.languages.locale`, [35](#)
- `dateparser.languages.validation`, [35](#)
- `dateparser.timezone_parser`, [39](#)
- `dateparser.timezones`, [40](#)
- `dateparser.utils`, [40](#)



## A

`apply_dateparser_timezone()` (in module `dateparser.utils`), 40  
`apply_settings()` (in module `dateparser.conf`), 37  
`apply_timezone()` (in module `dateparser.utils`), 40  
`apply_timezone_from_settings()` (in module `dateparser.utils`), 40  
`apply_tzdatabase_timezone()` (in module `dateparser.utils`), 40  
`are_tokens_valid()` (`dateparser.languages.dictionary.Dictionary` method), 33

## B

`build_tz_offsets()` (in module `dateparser.timezone_parser`), 39

## C

`CalendarBase` (class in `dateparser.calendars`), 36  
`check_settings()` (in module `dateparser.conf`), 37  
`clean_dictionary()` (`dateparser.languages.locale.Locale` static method), 35  
`combine_dicts()` (in module `dateparser.utils`), 40  
`convert_to_local_tz()` (in module `dateparser.timezone_parser`), 39  
`count_applicability()` (`dateparser.languages.locale.Locale` method), 35

## D

`date_range()` (in module `dateparser.date`), 38  
`DateData` (class in `dateparser.date`), 37  
`DateDataParser` (class in `dateparser.date`), 37  
`dateparser`  
 module, 40  
`DateParser` (class in `dateparser.date_parser`), 39  
`dateparser.calendars`  
 module, 36  
`dateparser.conf`  
 module, 36  
`dateparser.date`  
 module, 37  
`dateparser.date_parser`  
 module, 39

`dateparser.freshness_date_parser`  
 module, 39  
`dateparser.languages`  
 module, 36  
`dateparser.languages.dictionary`  
 module, 33  
`dateparser.languages.loader`  
 module, 34  
`dateparser.languages.locale`  
 module, 35  
`dateparser.languages.validation`  
 module, 35  
`dateparser.timezone_parser`  
 module, 39  
`dateparser.timezones`  
 module, 40  
`dateparser.utils`  
 module, 40  
`Dictionary` (class in `dateparser.languages.dictionary`), 33  
`dst()` (`dateparser.timezone_parser.StaticTzInfo` method), 39

## F

`find_date_separator()` (in module `dateparser.utils`), 40  
`FreshnessDateDataParser` (class in `dateparser.freshness_date_parser`), 39

## G

`get_date_data()` (`dateparser.date.DateDataParser` method), 37  
`get_date_data()` (`dateparser.freshness_date_parser.FreshnessDateDataParser` method), 39  
`get_date_from_timestamp()` (in module `dateparser.date`), 38  
`get_date_tuple()` (`dateparser.date.DateDataParser` method), 38  
`get_intersecting_periods()` (in module `dateparser.date`), 38  
`get_key()` (`dateparser.conf.Settings` class method), 37  
`get_kwargs()` (`dateparser.freshness_date_parser.FreshnessDateDataParser` method), 39

`get_last_day_of_month()` (in module `dateparser.timezone_parser`, 39  
`dateparser.utils`), 40  
`get_local_tz()` (`dateparser.freshness_date_parser.FreshnessDateParser`, 39  
`dateparser.utils`), 40  
`get_local_tz_offset()` (in module `dateparser.timezone_parser`), 39  
`get_locale()` (`dateparser.languages.loader.LocaleDataLoader`, 34  
`dateparser.languages.dictionary`), 33  
`get_locale_map()` (`dateparser.languages.loader.LocaleDataLoader`  
`method`), 34  
`get_locales()` (`dateparser.languages.loader.LocaleDataLoader`  
`method`), 34  
`get_logger()` (`dateparser.languages.validation.LanguageValidator`  
`class method`), 35  
`get_logger()` (in module `dateparser.utils`), 40  
`get_next_leap_year()` (in module `dateparser.utils`), 40  
`get_previous_leap_year()` (in module `dateparser.utils`), 40  
`get_timezone_from_tz_string()` (in module `dateparser.utils`), 40  
`get_wordchars_for_detection()` (`dateparser.languages.locale.Locale` `method`), 35

**I**

`is_applicable()` (`dateparser.languages.locale.Locale`  
`method`), 35

**L**

`LanguageValidator` (class in `dateparser.languages.validation`), 35  
`Locale` (class in `dateparser.languages.locale`), 35  
`locale_loader` (`dateparser.date.DateDataParser`  
`attribute`), 38  
`LocaleDataLoader` (class in `dateparser.languages.loader`), 34  
`localize()` (`dateparser.timezone_parser.StaticTzInfo`  
`method`), 39  
`localize_timezone()` (in module `dateparser.utils`), 40  
`logger` (`dateparser.languages.validation.LanguageValidator`  
`attribute`), 36

**M**

`module`  
`dateparser`, 40  
`dateparser.calendars`, 36  
`dateparser.conf`, 36  
`dateparser.date`, 37  
`dateparser.date_parser`, 39  
`dateparser.freshness_date_parser`, 39  
`dateparser.languages`, 36  
`dateparser.languages.dictionary`, 33  
`dateparser.languages.loader`, 34  
`dateparser.languages.locale`, 35  
`dateparser.languages.validation`, 35

**N**

`normalize_unicode()` (in module `dateparser.utils`), 40  
`NormalizedDictionary` (class in `dateparser.languages.dictionary`), 33

**P**

`parse()` (`dateparser.date_parser.DateParser` `method`), 39  
`parse()` (`dateparser.freshness_date_parser.FreshnessDateParser`  
`method`), 39  
`parse()` (in module `dateparser`), 40  
`parse_with_formats()` (in module `dateparser.date`), 38  
`pop_tz_offset_from_string()` (in module `dateparser.timezone_parser`), 39

**R**

`registry()` (in module `dateparser.utils`), 40  
`replace()` (`dateparser.conf.Settings` `method`), 37

**S**

`sanitize_date()` (in module `dateparser.date`), 39  
`sanitize_spaces()` (in module `dateparser.date`), 39  
`set_correct_day_from_settings()` (in module `dateparser.utils`), 40  
`set_correct_month_from_settings()` (in module `dateparser.utils`), 40  
`Settings` (class in `dateparser.conf`), 36  
`SettingValidationError`, 36  
`setup_logging()` (in module `dateparser.utils`), 40  
`split()` (`dateparser.languages.dictionary.Dictionary`  
`method`), 33  
`StaticTzInfo` (class in `dateparser.timezone_parser`), 39  
`strip_braces()` (in module `dateparser.utils`), 40

**T**

`to_parserinfo()` (`dateparser.languages.locale.Locale`  
`method`), 35  
`translate()` (`dateparser.languages.locale.Locale`  
`method`), 35  
`translate_search()` (`dateparser.languages.locale.Locale`  
`method`), 35  
`tzname()` (`dateparser.timezone_parser.StaticTzInfo`  
`method`), 39

**U**

`UnknownTokenError`, 33  
`utcoffset()` (`dateparser.timezone_parser.StaticTzInfo`  
`method`), 39

## V

`VALID_KEYS` (*dateparser.languages.validation.LanguageValidator*  
*attribute*), [35](#)

`validate_info()` (*dateparser.languages.validation.LanguageValidator*  
*class method*), [36](#)

## W

`word_is_tz()` (*in module dateparser.timezone\_parser*),  
[39](#)