DateParser Documentation

Release 0.6.0

Scrapinghub

Mar 13, 2017

Contents

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

CHAPTER 1

Documentation

Documentation is built automatically and can be found on Read the Docs.

CHAPTER 2

Features

- Generic parsing of dates in English, Spanish, Dutch, Russian and over 20 other languages 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'`.
- Support for non-Gregorian calendar systems. See *Supported Calendars*.
- Extensive test coverage.

CHAPTER 3

Usage

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

`dateparser.`**`parse`**`(`*`*args`*`, `*`**kwargs`*`)`

Parse date and time from given date string.

Parameters

- **`date_string`** (*`str|unicode`*) – 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.
- **`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`.

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

Return type `datetime`.

Raises ValueError - Unknown Language

Popular Formats

```
>>> import dateparser
>>> dateparser.parse('12/12/12')
datetime.datetime(2012, 12, 12, 0, 0)
>>> dateparser.parse(u'Fri, 12 Dec 2014 10:55:50')
datetime.datetime(2014, 12, 12, 10, 55, 50)
>>> dateparser.parse(u'Martes 21 de Octubre de 2014')  # Spanish (Tuesday 21 October␣
↪2014)
```

```
datetime.datetime(2014, 10, 21, 0, 0)
>>> dateparser.parse(u'Le 11 Décembre 2014 à 09:00')  # French (11 December 2014 at␣
↪09:00)
datetime.datetime(2014, 12, 11, 9, 0)
>>> dateparser.parse(u'13  2015 .  13:34')  # Russian (13 January 2015 at 13:34)
datetime.datetime(2015, 1, 13, 13, 34)
>>> dateparser.parse(u'1  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(u'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(u'Il ya 2 heures')  # French (2 hours ago)
datetime.datetime(2015, 5, 31, 22, 0)
>>> parse(u'1 anno 2 mesi')  # Italian (1 year 2 months)
datetime.datetime(2014, 4, 1, 0, 0)
>>> parse(u'yaklaşık 23 saat önce')  # Turkish (23 hours ago)
datetime.datetime(2015, 5, 31, 1, 0)
>>> parse(u'Hace una semana')  # Spanish (a week ago)
datetime.datetime(2015, 5, 25, 0, 0)
>>> parse(u'2')  # 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: Support for relative dates in future needs a lot of improvement, we look forward to community's contribution to get better on that part. See **'Contributing'_**.

OOTB Language Based Date Order Preference

```
>>> # parsing ambiguous date
>>> parse('02-03-2016')  # assumes english language, uses MDY date order
datetime.datetime(2016, 3, 2, 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 usings *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(u'December 2015')  # default behavior
datetime.datetime(2015, 12, 16, 0, 0)
>>> parse(u'December 2015', settings={'PREFER_DAY_OF_MONTH': 'last'})
datetime.datetime(2015, 12, 31, 0, 0)
>>> parse(u'December 2015', settings={'PREFER_DAY_OF_MONTH': 'first'})
datetime.datetime(2015, 12, 1, 0, 0)
```

```
>>> parse(u'March')
datetime.datetime(2015, 3, 16, 0, 0)
>>> parse(u'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)
```

You can also ignore parsing incomplete dates altogether by setting *STRICT_PARSING* flag as follows:

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

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

CHAPTER 4

Dependencies

dateparser relies on following libraries in some ways:

- dateutil's module `relativedelta` for its freshness parser.
- ruamel.yaml for reading language and configuration files.
- jdatetime to convert *Jalali* dates to *Gregorian*.
- umalqurra to convert *Hijri* dates to *Gregorian*.
- tzlocal to reliably get local timezone.

CHAPTER 5

Supported languages

- Arabic
- Bangla
- Belarusian
- Bulgarian
- Chinese
- Czech
- Danish
- Dutch
- English
- Filipino/Tagalog
- Finnish
- French
- Hebrew
- Hindi
- Hungarian
- German
- Indonesian
- Italian
- Japanese
- Persian
- Polish
- Portuguese

- Romanian
- Russian
- Spanish
- Thai
- Turkish
- Ukrainian
- Vietnamese

CHAPTER 6

Supported Calendars

- Gregorian calendar.
- Persian Jalali calendar. For more information, refer to Persian Jalali Calendar.
- Hijri/Islamic Calendar. For more information, refer to Hijri Calendar.

```
>>> from dateparser.calendars.jalali import JalaliCalendar
>>> JalaliCalendar(u'    ').get_date()
{'date_obj': datetime.datetime(2009, 3, 20, 0, 0), 'period': 'day'}
```

```
>>> from dateparser.calendars.hijri import HijriCalendar
>>> HijriCalendar(u'17-01-1437  08:30 ').get_date()
{'date_obj': datetime.datetime(2015, 10, 30, 20, 30), 'period': 'day'}
```

Note: *HijriCalendar* has some limitations with Python 3.

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

Install using following command to use calendars.

Tip: pip install dateparser[calendars]

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` reduces the number of applicable languages, until only one or no language is left. It assumes the previously detected language for all the subsequent dates supplied.

This class wraps around the core *dateparser* functionality, and by default assumes that all of the dates fed to it are in the same language.

class `dateparser.date.`**DateDataParser**(**args*, ***kwargs*)

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

Parameters

- **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.
- **allow_redetect_language** (*bool*) – Enables/disables language re-detection.
- **settings** (*dict*) – Configure customized behavior using settings defined in `dateparser.conf.Settings`.

Returns A parser instance

Raises ValueError - Unknown Language, TypeError - Languages argument must be a list

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|unicode*) – 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 dict mapping keys to `datetime.datetime` object and *period*. For example: {'date_obj': datetime.datetime(2015, 6, 1, 0, 0), 'period': u'day'}

Raises ValueError - Unknown Language

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

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(u'March 2015')
{'date_obj': datetime.datetime(2015, 3, 16, 0, 0), 'period': u'month'}
```

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(u'2014')
{'date_obj': datetime.datetime(2014, 6, 16, 0, 0), 'period': u'year'}
```

Dates with time zone indications or UTC offsets are returned in UTC time unless specified using *Settings*.

```
>>> DateDataParser().get_date_data(u'23 March 2000, 1:21 PM CET')
{'date_obj': datetime.datetime(2000, 3, 23, 14, 21), 'period': 'day'}
```

Warning: It fails to parse *English* dates in the example below, because *Spanish* was detected and stored with the `ddp` instance:

```
>>> ddp.get_date_data('11 August 2012')
{'date_obj': None, 'period': 'day'}
```

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

```
>>> ddp = DateDataParser(languages=['de', 'nl'])
>>> ddp.get_date_data(u'vr jan 24, 2014 12:49')
{'date_obj': datetime.datetime(2014, 1, 24, 12, 49), 'period': u'day'}
>>> ddp.get_date_data(u'18.10.14 um 22:56 Uhr')
{'date_obj': datetime.datetime(2014, 10, 18, 22, 56), 'period': u'day'}
```

CHAPTER 7

Settings

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

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

`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_LANGUAGE_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, 3, 2, 0, 0)
>>> parse('le 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_LANGUAGE_DATE_ORDER* to *False*:

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

```
>>> parse('le 02-03-2016', settings={'DATE_ORDER': 'MDY', 'PREFER_LANGUAGE_DATE_ORDER
↪': False})
datetime.datetime(2016, 2, 3, 0, 0)  # MDY worked!
```

`TIMEZONE` defaults to local timezone. When specified, resultant `datetime` is localized with the given timezone.

```
>>> 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` is a flag to toggle between timezone aware/naive dates:

```
>>> 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)
```

`PREFER_DAY_OF_MONTH` This option comes handy when the date string is missing the day part. It defaults to `current` and can have `first` and `last` denoting first and last day of months respectively as values:

```
>>> from dateparser import parse
>>> parse(u'December 2015')  # default behavior
datetime.datetime(2015, 12, 16, 0, 0)
>>> parse(u'December 2015', settings={'PREFER_DAY_OF_MONTH': 'last'})
datetime.datetime(2015, 12, 31, 0, 0)
>>> parse(u'December 2015', settings={'PREFER_DAY_OF_MONTH': 'first'})
datetime.datetime(2015, 12, 1, 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(u'March')
datetime.datetime(2015, 3, 16, 0, 0)
>>> parse(u'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(u'14:30')
datetime.datetime(2015, 3, 16, 14, 30)
```

```
>>> parse(u'14:30', settings={'RELATIVE_BASE': datetime.datetime(2020, 1, 1)})
datetime.datetime(2020, 1, 1, 14, 30)
>>> parse(u'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(u'March', settings={'STRICT_PARSING': True})
None
```

`SKIP_TOKENS` 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)
{'date_obj': datetime.datetime(1981, 6, 27, 0, 0), 'period': 'day'}
```

CHAPTER 8

Documentation

Contents:

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
```

Deploying dateparser in a Scrapy Cloud project

The initial use cases for *dateparser* were for Scrapy projects doing web scraping that needed to parse dates from websites. These instructions show how you can deploy it in a Scrapy project running in Scrapy Cloud.

Deploying with shub

The most straightforward way to do that is to use the latest version of the shub command line tool.

First, install `shub`, if you haven't already:

```
pip install shub
```

Then, you can choose between deploying a stable release or the latest from development.

Deploying a stable dateparser release:

1. Then, use `shub` to install python-dateutil (we require at least 2.3 version), jdatetime and PyYAML dependencies from PyPI:

```
shub deploy-egg --from-pypi python-dateutil YOUR_PROJECT_ID
shub deploy-egg --from-pypi jdatetime YOUR_PROJECT_ID
shub deploy-egg --from-pypi PyYAML YOUR_PROJECT_ID
```

2. Finally, deploy dateparser from PyPI:

```
shub deploy-egg --from-pypi dateparser YOUR_PROJECT_ID
```

Deploying from latest sources

Optionally, you can deploy it from the latest sources:

Inside the `dateparser` root directory:

1. Run the command to deploy the dependencies:

```
shub deploy-reqs YOUR_PROJECT_ID requirements.txt
```

2. Then, either deploy from the latest sources on GitHub:

```
shub deploy-egg --from-url git@github.com:scrapinghub/dateparser.git YOUR_PROJECT_
↪ID
```

Or, just deploy from the local sources (useful if you have local modifications):

```
shub deploy-egg
```

Deploying the egg manually

In case you run into trouble with the above procedure, you can deploy the egg manually. First clone the `dateparser`'s repo, then inside its directory run the command:

```
python setup.py bdist_egg
```

After that, you can upload the egg using Scrapy Cloud's Dashboard interface under Settings > Eggs section.

Dependencies

Similarly, you can download source and package PyYAML, jdatetime and dateutil (version >= 2.3) as *eggs* and deploy them like above.

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

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it. We encourage you to add new languages to 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.

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:

```
$ pip install -r tests/requirements.txt # install test dependencies
$ flake8 dateparser tests
$ nosetests
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv. (Note that we use `max-line-length = 100` for flake8, this is configured in `setup.cfg` file.)

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 https://travis-ci.org/scrapinghub/dateparser/pull_requests and make sure that the tests pass for all supported Python versions.
4. Follow the core developers' advice which aim to ensure code's consistency regardless of variety of approaches used by many contributors.
5. 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 Adding New Languages

English is the primary language of the dateparser. Dates in all other languages are translated into English equivalents before they are parsed. The language data required for parsing dates is contained in *data/languages.yml* file. It

contains variable parts that can be used in dates, language by language: month and week names - and their abbreviations, prepositions, conjunctions and frequently used descriptive words and phrases (like "today"). The chosen data format is YAML because it is readable and simple to edit. Language data is extracted per language from YAML with `LanguageDataLoader` and validated before being put into `Language` class.

Refer to language-data-template for details about its structure and take a look at already implemented languages for examples. As we deal with the delicate fabric of interwoven languages, tests are essential to keep the functionality across them. Therefore any addition or change should be reflected in tests. However, there is nothing to be afraid of: our tests are highly parameterized and in most cases a test fits in one declarative line of data. Alternatively, you can provide required information and ask the maintainers to create the tests for you.

Credits

Committers

- Adam LeVasseur
- Alec Koumjian
- Ammar Azif
- Artur Sadurski
- Artur Gaspar
- Benjamin Bach
- Claudio Salazar
- Cesar Flores
- conanca
- David Beitey
- demelziraptor
- Elias Dorneles
- Eugene Amirov
- Faisal Anees
- Ismael Carnales
- Jolo Balbin
- Joseph Kahn
- Mark Baas
- Marko Horvatić
- Mateusz Golewski
- Michael Palumbo
- Opp Lieamsiriwong
- Paul Tremberth
- Rajat Goyal
- Raul Gallegos

- Roman
- Shuai Lin
- Sigit Dewanto
- Sviatoslav Sydorenko
- Taito Horiuchi
- Takahiro Kamatani
- Thomas Steinacher
- Tom Russell
- Umair Ashraf
- Waqas Shabir

History

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 langauge.
- Added setting, *STRICT_PARSING* to ignore imcomplete 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 Ukranian 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.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

D

G

P