DateParser Documentation

Release 0.3.1

Scrapinghub

October 28, 2015

Contents

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

CHAPTER 1

Documentation

Documentation can be found here.

CHAPTER 2

Features

- Generic parsing of dates in English, Spanish, Dutch, Russian and several other languages and formats.
- Generic parsing of relative dates like: `'1 min ago'`,`'2 weeks ago'`,`'3 months, 1 week and 1 day ago'`.
- 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 with the first addition of `JalaliParser`. See Persian Jalali Calendar for more information.
- 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`**(*date_string*, *date_formats=None*, *languages=None*)

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.

Returns Returns a `datetime.datetime` if successful, else returns None

Raises ValueError - Unknown Language

3.1 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)
```

3.2 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.

CHAPTER 4

Dependencies

dateparser translates non-English dates to English and uses dateutil module `parser` to parse the translated date.

Also, it requires PyYAML for its language detection module to work. The module jdatetime is used for handling Jalali calendar.

Supported languages

- Arabic
- Belarusian
- Chinese
- Czech
- Dutch
- English
- Filipino
- French
- German
- Indonesian
- Italian
- Persian
- Polish
- Portuguese
- Romanian
- Russian
- Spanish
- Thai
- Turkish
- Ukrainian
- Vietnamese

CHAPTER 6

Supported Calendars

- Gregorian calendar
- Persian Jalali calendar

CHAPTER 7

Example of Use for Jalali Calendar

```
>>> from dateparser.calendars.jalali import JalaliParser
>>> JalaliParser(u'    ').get_date()
datetime.datetime(2009, 3, 20, 0, 0)
```

7.1 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.

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

The instance of `dateparser.date.DateDataParser` reduces the number of applicable languages, until only one or no language is left. It assumes the previously detected language for all the next dates and does not try to execute the language detection again after a language is discarded.

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**(*languages=None*, *allow_redetect_language=False*)

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.

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.

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

Once initialized, `dateparser.date.DateDataParser.get_date_data()` parses date strings:

```
>>> from dateparser.date import DateDataParser
>>> ddp = DateDataParser()
>>> ddp.get_date_data(u'Martes 21 de Octubre de 2014')  # Spanish
{'date_obj': datetime.datetime(2014, 10, 21, 0, 0), 'period': u'day'}
>>> ddp.get_date_data(u'13 Septiembre, 2014')  # Spanish
{'date_obj': datetime.datetime(2014, 9, 13, 0, 0), 'period': u'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 8

Documentation

Contents:

8.1 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
```

8.1.1 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.

8.2 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:

8.2.1 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 :)

8.2.2 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.

8.2.3 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.

8.2.4 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.

8.3 Credits

8.3.1 Committers

- Artur Sadurski
- Claudio Salazar
- Cesar Flores
- Elias Dorneles
- Eugene Amirov
- Faisal Anees
- Ismael Carnales
- Jolo Balbin
- Joseph Kahn
- Mark Baas
- Marko Horvatić
- Mateusz Golewski
- Opp Lieamsiriwong
- Rajat Goyal
- Raul Gallegos
- Shuai Lin
- Sigit Dewanto
- Sviatoslav Sydorenko
- Tom Russell
- Umair Ashraf
- Waqas Shabir

8.4 History

8.4.1 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.

8.4.2 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.

8.4.3 0.2.1 (2015-07-13)

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

8.4.4 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.

8.4.5 0.1.0 (2014-11-24)

- First release on PyPI.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

d

D

G

P