TreeTagger Python Wrapper Documentation

Release 2.3

Laurent Pointal

Jan 29, 2019

Contents

CHAPTER 1

About treetaggerwrapper

author Laurent Pointal <laurent.pointal@limsi.fr> <laurent.pointal@laposte.net>

organization CNRS - LIMSI

version 2.3

For language independent part-of-speech tagger TreeTagger, see Helmut Schmid TreeTagger site.

For this module, see Developer Project page and Project Source repository on french academic repository SourceSup. And Module Documentation on Read The Docs.

You can also retrieve the latest version of this module with the svn command:

```
svn export https://subversion.renater.fr/ttpw/trunk/treetaggerwrapper.py
```

Or install it (and the module treetaggerpoll.py) using pip (add pip install option `--user` for user private installation):

```
pip install treetaggerwrapper
```

This wrapper tool is intended to be used in projects where multiple chunk of texts must be processed via TreeTagger in an automatic way (else you may simply use the base TreeTagger installation once as an external command).

Warning: Parameter files renaming.

Latest distributed files on TreeTagger site removed `-utf8` part from parameter files names. This version 2.3 ot the wrapper tries to adapt to your installed version of TreeTagger: test existence of `.par` file without `-utf8` part, and if it failed, test existence of file with adding `-utf8` part.

If you use this wrapper, a small email would be welcome to support module maintenance (where, purpose, funding…). Send it to laurent.pointal@limsi.fr

CHAPTER 2

Installation

2.1 Requirements

`treetaggerwrapper` rely on `six` module for Python2 and Python3 compatibility. It also uses standard `io` module for files reading with decoding / encoding .

Tests have been limited to Python 2.7 and Python 3.4 under Linux and Windows. It don't work with earlier version of Python as some names are not defined in their standard libraries.

2.2 Automatic

As the module is now registered on PyPI, you can simply install it:

```
pip install treetaggerwrapper
```

Or, if you can't (or don't want) to install the module system-wide (and don't use a virtual env):

```
pip install --user treetaggerwrapper
```

May use `pip3` to go with your Python3 installation.

If it is already installed as a package, use pip's install `-U` option to install the last version (update).

2.3 Manual

For a complete manual installation, install `six` module and other dependencies, and simply put the `treetaggerwrapper.py` and `treetaggerpoll.py` files in a directory listed in the Python path (or in your scripts directory).

CHAPTER 3

Configuration

The wrapper search for the treetagger directory (the one with `bin`, `lib` and `cmd` subdirectories), in several places, allowing variations in TreeTagger directory name — see TreeTagger automatic locate for details.

If the treetagger directory is found, its location is stored in a file `$HOME/.config/treetagger_wrapper.cfg` (or any place following `XDG_CONFIG_DIR` if it is specified), and at next start the directory indicated in this file is used if it still exists.

If you installed TreeTagger in a non-guessable location, you still can set up an environment variable `TAGDIR` to reference the TreeTagger software installation directory, or give a *TAGDIR* named argument when building a `TreeTagger` object to provide this information, or simply put that information into configuration file in section `[CACHE]` under key `tagdir = ...`.

CHAPTER 4

Usage

Primary usage is to wrap TreeTagger binary and use it as a functional tool. You have to build a *TreeTagger* object, specifying the target language [by its country code!], and possibly some other TreeTagger parameters (else we use standard files specified in the module for each supported language). Once this wrapper object created, you can simply call its *tag_text()* method with the string to tag, and it will return a list of lines corresponding to the text tagged by TreeTagger.

Example (with Python3, **Unicode strings** by default — with Python2 you need to use explicit notation `u"string"`, of if within a script start by a `from __future__ import unicode_literals` directive):

```
>>> import pprint   # For proper print of sequences.
>>> import treetaggerwrapper
>>> #1) build a TreeTagger wrapper:
>>> tagger = treetaggerwrapper.TreeTagger(TAGLANG='en')
>>> #2) tag your text.
>>> tags = tagger.tag_text("This is a very short text to tag.")
>>> #3) use the tags list... (list of string output from TreeTagger).
>>> pprint.pprint(tags)
['This\tDT\tthis',
 'is\tVBZ\tbe',
 'a\tDT\ta',
 'very\tRB\tvery',
 'short\tJJ\tshort',
 'text\tNN\ttext',
 'to\tTO\tto',
 'tag\tVV\ttag',
 '.\tSENT\t.']
>>> # Note: in output strings, fields are separated with tab chars (\t).
```

You can transform it into a list of named tuples `Tag`, `NotTag` (unknown tokens) `TagExtra` (token having extra informations requested via tagger options - like probabilistic indications) using the helper *make_tags()* function:

```
>>> tags2 = treetaggerwrapper.make_tags(tags)
>>> pprint.pprint(tags2)
[Tag(word='This', pos='DT', lemma='this'),
```

(continues on next page)

(continued from previous page)

```
 Tag(word='is', pos='VBZ', lemma='be'),
 Tag(word='a', pos='DT', lemma='a'),
 Tag(word='very', pos='RB', lemma='very'),
 Tag(word='short', pos='JJ', lemma='short'),
 Tag(word='text', pos='NN', lemma='text'),
 Tag(word='to', pos='TO', lemma='to'),
 Tag(word='tag', pos='VV', lemma='tag'),
 Tag(word='.', pos='SENT', lemma='.')]
```

You can also directly process files using *TreeTagger.tag_file()* and *TreeTagger.tag_file_to()* methods.

The module itself can be used as a command line tool too, for more information ask for module help:

```
python treetaggerwrapper.py --help
```

If available within `PYTHONPATH`, the module can also be called from anywhere with the `-m` Python option:

```
python -m treetaggerwrapper --help
```

CHAPTER 5

Important modifications notes

On august 2015, the module has been reworked deeply, some modifications imply modifications in users code.

- **Methods renamed** (and functions too) to follow Python rules, they are now lowercase with underscore separator between words. Typically for users, `tt.TagText()` becomes `tt.tag_text()` (for this method a compatibility method has been written, but no longer support lists of non-Unicode strings).
- Work with Python2 and Python3, with same code.
- Use **Unicode strings** internally (it's no more possible to provide binary strings and their encoding as separated parameters - you have to decode the strings yourself before calling module functions).
- Assume **utf-8** when dealing with *TreeTagger binary*, default to its utf-8 versions of parameter and abbrev files. If you use another encoding, you must specify these files: in your sources, or via environment vars, or in the `treetagger_wrapper.cfg` configuration file under encoding name section (respecting Python encoding names as given by `codecs.lookup(enc).name`, ie. uses `utf-8`).
- Default to **utf-8** when reading *user files* (you need to specify latin1 if you use such encoding - previously it was the default).
- **Guess TreeTagger location** — you can still provide `TAGDIR` as environment variable or as *`TreeTagger`* parameter, but it's no more necessary. Found directory is cached in `treetagger_wrapper.cfg` configuration file to only guess once.
- Documentation has been revised to only export main things for module usage; internals stay documented via comments in the source.
- **Text chunking** (tokenizing to provide treetagger input) has been revisited and must be more efficient. And you can now also provide your own external chunking function when creating the wrapper — which will replace internal chunking in the whole process.
- XML tags generated have been modified (made shorted and with `ttpw:` namespace).
- Can be used in **multithreading** context (pipe communications with TreeTagger are protected by a Lock, preventing concurrent access). If you need multiple parallel processing, you can create multiple *`TreeTagger`* objects, put them in a poll, and work with them from different threads.
- Support polls of taggers for optimal usage on multi-core computers. See *`treetaggerwrapper.TaggerPoll`* class for thread poll and *`treetaggerpoll.TaggerProcessPoll`* class for process poll.

CHAPTER 6

Processing

6.1 This module does two main things

- Manage preprocessing of text (chunking to extract tokens for treetagger input) in place of external Perl scripts as in base TreeTagger installation, thus avoid starting Perl each time a piece of text must be tagged.
- Keep alive a pipe connected to TreeTagger process, and use that pipe to send data and retrieve tags, thus avoid starting TreeTagger each time and avoid writing / reading temporary files on disk (direct communication via the pipe). Ensure flushing of tagger output.

6.1.1 Supported languages

Note: Encoding specification

When specifying language with treetaggerwrapper, we use the the two chars language codes, not the complete language name.

This module support chunking (tokenizing) + tagging for languages:

- spanish (es)
- french (fr)
- english (en)
- german (de)

It can be used for tagging only for languages:

- bulgarian (bg)
- dutch (nl)
- estonian (et)

- finnish (fi)
- galician (gl)
- italian (it)
- korean (kr)
- latin (la)
- mongolian (mn)
- polish (pl)
- russian (ru)
- slovak (sk')
- swahili (sw)

Note: chunking parameters have not been adapted to these language and their specific features, you may try to chunk with default processing… with no guaranty. If you have an external chunker, you can call the tagger with option `tagonly` set to True, you should then provide a simple string with one token by line (or list of strings with one token by item). If you chunker is a callable, you can provide your own chunking function with `CHUNKERPROC` named parameter when constructing *`TreeTagger`* object, and then use it normally (your function is called in place of standard chunking).

For all these languages, the wrapper use standard filenames for TreeTagger's parameter and abbreviation files. You can override these names using `TAGPARFILE` and `TAGABBREV` parameters, and then use alternate files.

6.2 Other things done by this module

- Can number lines into XML tags (to identify lines after TreeTagger processing).
- Can mark whitespaces with XML tags.
- By default replace non-talk parts like URLs, emails, IP addresses, DNS names (can be turned off). Replaced by a 'replaced-xxx' string followed by an XML tag containing the replaced text as attribute (the tool was originally used to tag parts of exchanges from technical mailing lists, containing many of these items).
- Acronyms like U.S.A. are systematically written with a final dot, even if it is missing in original file.
- Automatic encode/decode files using user specified encoding (default to utf-8).

In normal mode, all journal outputs are done via Python standard logging system, standard output is only used if a) you run the module in pipe mode (ie. results goes to stdout), or b) you set DEBUG or DEBUG_PREPROCESS global variables and you use the module directly on command line (which make journal and other traces to be sent to stdout).

For an example of logging use, see *`enable_debugging_log()`* function.

6.3 Alternative tool

You may also take a look at project treetagger python which wraps TreeTagger command-line tools (simpler than this module, it may be slower if you have many texts to tag in your process as it calls and restarts TreeTagger chunking then tagging tools chain for each text).

CHAPTER 7

Hints

7.1 Window buffer overflow

On windows, if you get the following error about some file manipulation (ex. in an `abspath()` call):

```
TypeError: must be (buffer overflow), not str
```

Check that directories and filenames total length don't exceed 260 chars. If this is the case, you may try to use UNC names starting by `\\?\` (read Microsoft Naming Files, Paths, and Namespaces documentation — note: you cannot use `/` to separate directories with this notation).

7.2 TreeTagger automatic location

For your TreeTagger to be automatically find by the script, its **directory** must follow installation rules below:

7.2.1 Directory naming and content

Location search function tries to find a directory beginning with `tree`, possibly followed by any char (ex. a space, a dash…), followed by `tagger`, possibly followed by any sequence of chars (ex. a version number), and without case distinction.

This match directory names like `treetagger`, `TreeTagger`, `Tree-tagger`, `Tree Tagger`, `treetagger-2.0`…

The directory must contain `bin` and `lib` subdirectories (they are normally created by TreeTagger installation script, or directly included in TreeTagger Windows zipped archive).

First directory corresponding to these criteria is considered to be the TreeTagger installation directory.

7.2.2 Searched locations

TreeTagger directory location is searched from local (user private installation) to global (system wide installation).

1. Near the `treetaggerwrapper.py` file (TreeTagger being in same directory).
2. Containing the `treetaggerwraper.py` file (module inside TreeTagger directory).
3. User home directory (ex. `/home/login`, `C:\Users\login`).
4. First level directories in user home directory (ex. `/home/login/tools`, `C:\Users\login\Desktop`).
5. For MacOSX, in `~/Library/Frameworks`.
6. For Windows, in program files directories (ex. `C:\Program Files`).
7. For Windows, in each existing fixed disk root and its first level directories (ex. `C:\`, `C:\Tools`, `E:\`, `E:\Apps`).
8. For Posix (Linux, BSD… MacOSX), in a list of standard directories:
   - `/usr/bin`,
   - `/usr/lib`,
   - `/usr/local/bin`,
   - `/usr/local/lib`,
   - `/opt`,
   - `/opt/bin`,
   - `/opt/lib`,
   - `/opt/local/bin`,
   - `/opt/local/lib`.
9. For MacOSX, in applications standard directories:
   - `/Applications`,
   - `/Applications/bin`,
   - `/Library/Frameworks`.

7.3 TreeTagger probabilities

Using `TAGOPT` parameter when constructing *`TreeTagger`* object, you can provide `-threshold` and `-prob` parameters to the treetagger process, and then retrieve probability informations in the tagger output (see TreeTagger README file for all options).

```
>>> import treetaggerwrapper as ttpw
>>> tagger = ttpw.TreeTagger(TAGLANG='fr', TAGOPT="-prob -threshold 0.7 -token -lemma␣
↪-sgml -quiet")
>>> tags = tagger.tag_text('Voici un petit test de TreeTagger pour voir.')
>>> import pprint
>>> pprint.pprint(tags)
['Voici\tADV voici 1.000000',
'un\tDET:ART un 0.995819',
'petit\tADJ petit 0.996668',
'test\tNOM test 1.000000',
```

(continues on next page)

(continued from previous page)

```
'de\tPRP de 1.000000',
'TreeTagger\tNAM <unknown> 0.966699',
'pour\tPRP pour 0.663202',
'voir\tVER:infi voir 1.000000',
'.\tSENT . 1.000000']
>>> tags2 = ttpw.make_tags(tags, allow_extra=True)
>>> pprint.pprint(tags2)
[TagExtra(word='Voici', pos='ADV', lemma='voici', extra=(1.0,)),
TagExtra(word='un', pos='DET:ART', lemma='un', extra=(0.995819,)),
TagExtra(word='petit', pos='ADJ', lemma='petit', extra=(0.996668,)),
TagExtra(word='test', pos='NOM', lemma='test', extra=(1.0,)),
TagExtra(word='de', pos='PRP', lemma='de', extra=(1.0,)),
TagExtra(word='TreeTagger', pos='NAM', lemma='<unknown>', extra=(0.966699,)),
TagExtra(word='pour', pos='PRP', lemma='pour', extra=(0.663202,)),
TagExtra(word='voir', pos='VER:infi', lemma='voir', extra=(1.0,)),
TagExtra(word='.', pos='SENT', lemma='.', extra=(1.0,))]
```

Note: This provides extra data for each token, your script must be adapted for this (you can note in the pprint formated display that we have tab and space separators — a tab after the word, then spaces between items).

CHAPTER 8

Module exceptions, class and functions

exception `treetaggerwrapper.`**`TreeTaggerError`**

For exceptions generated directly by TreeTagger wrapper.

class `treetaggerwrapper.`**`TreeTagger`**`(`***kargs*`)`

Wrap TreeTagger binary to optimize its usage on multiple texts.

The two main methods you may use are the `__init__()` initializer, and the `tag_text()` method to process your data and get TreeTagger output results.

Construction of a wrapper for a TreeTagger process.

You can specify several parameters at construction time. These parameters can be set via environment variables too (except for CHUNKERPROC). All of them have standard default values, even TAGLANG default to tagging english.

Parameters

- **TAGLANG** (*string*) – language code for texts ('en','fr',. . .) (default to 'en').
- **TAGDIR** (*string*) – path to TreeTagger installation directory.
- **TAGOPT** (*string*) – options for TreeTagger (default to '-token -lemma -sgml -quiet', it is recomanded to **keep these default options** for correct use of this tool, and add other options on your need).
- **TAGPARFILE** (*string*) – parameter file for TreeTagger. (default available for supported languages). Use value None to force use of default if environment variable define a value you don't wants to use.
- **TAGABBREV** (*string*) – abbreviation file for preprocessing. (default available for supported languages).
- **TAGINENC** (*str*) – encoding to use for TreeTagger input, default to utf8.
- **TAGOUTENC** (*str*) – encoding to use for TreeTagger output, default to utf8
- **TAGINENCERR** (*str*) – management of encoding errors for TreeTagger input, strict or ignore or replace - default to replace.

- **TAGOUTENCERR** (*str*) – management of encoding errors for TreeTagger output, strict or ignore or replace - default to replace.
- **CHUNKERPROC** (*fct(tagger, ['text']) => list ['chunk']*) – function to call for chunking in place of wrapper's chunking — default to None (use standard chunking). Take the TreeTagger object as first parameter and a list of str to chunk as second parameter. Must return a list of chunk str (tokens). Note that normal initialization of chunking parameters is done even with an external chunking function, so these parameters are available for this function.

Returns None

tag_text (*text*, *numlines=False*, *tagonly=False*, *prepronly=False*, *tagblanks=False*, *notagurl=False*, *notagemail=False*, *notagip=False*, *notagdns=False*, *nosgmlsplit=False*)
Tag a text and returns corresponding lines.

This is normally the method you use on this class. Other methods are only helpers of this one.

The return value of this method can be processed by `make_tags()` to retrieve a list of `Tag` named tuples with meaning fields.

Parameters

- **text** (*unicode string / [unicode string]*) – the text to tag.
- **numlines** (*boolean*) – indicator to keep line numbering information in data flow (done via SGML tags) (default to False).
- **tagonly** (*boolean*) – indicator to only do TreeTagger tagging processing on input (default to False). If tagonly is set, providen text must be composed of one token by line (either as a collection of line-feed separated lines in one string, or as a list of lines).
- **prepronly** (*boolean*) – indicator to only do preprocessing of text without tagging (default to False).
- **tagblanks** (*boolean*) – indicator to keep blanks characters information in data flow (done via SGML tags) (default to False).
- **notagurl** (*boolean*) – indicator to not do URL replacement (default to False).
- **notagemail** (*boolean*) – indicator to not do email address replacement (default to False).
- **notagip** (*boolean*) – indicator to not do IP address replacement (default to False).
- **notagdns** (*boolean*) – indicator to not do DNS names replacement (default to False).
- **nosgmlsplit** (*boolean*) – indicator to not split on sgml already within the text (default to False).

Returns List of output strings from the tagger. You may use `make_tags()` function to build a corresponding list of named tuple, for further processing readbility.

Return type [str]

tag_file (*infilepath*, *encoding='utf-8'*, *numlines=False*, *tagonly=False*, *prepronly=False*, *tagblanks=False*, *notagurl=False*, *notagemail=False*, *notagip=False*, *notagdns=False*, *nosgmlsplit=False*)
Call `tag_text()` on the content of a specified file.

Parameters

- **infilepath** (*str*) – pathname to access the file to read.
- **encoding** (*str*) – specify encoding of the file to read, default to utf-8.

Returns List of output strings from the tagger.

Return type [str]

Other parameters are simply passed to `tag_text()`.

`tag_file_to`(*infilepath*, *outfilepath*, *encoding='utf-8'*, *numlines=False*, *tagonly=False*, *prepronly=False*, *tagblanks=False*, *notagurl=False*, *notagemail=False*, *notagip=False*, *notagdns=False*, *nosgmlsplit=False*)

Call `tag_text()` on the content of a specified file and write result to a file.

Parameters

- **`infilepath`** (*`str`*) – pathname to access the file to read.
- **`outfilepath`** (*`str`*) – pathname to access the file to write.
- **`encoding`** (*`str`*) – specify encoding of the files to read/write, default to utf-8.

Other parameters are simply passed to `tag_text()`.

`treetaggerwrapper.`**`make_tags`**(*result*, *exclude_nottags=False*, *allow_extra=False*)

Tool function to transform a list of TreeTagger tabbed text output strings into a list of `Tag`/`TagExtra`/`NotTag` named tuples.

You call this function using the result of a `TreeTagger.tag_text()` call. `Tag` and `TagExtra` have attributes `word`, `pos` and `lemma`. `TagExtra` has an `extra` attribute containing a tuple of tagger's output complement values (where numeric values are converted to float). `NotTag` has a simple attribute `what`.

Parameters

- **`result`** – result of a `TreeTagger.tag_text()` call.
- **`exclude_nottags`** (*`bool`*) – dont generate `NotTag` for wrong size outputs. Default to False.
- **`allow_extra`** (*`bool`*) – build a `TagExtra` for outputs longer than expected. Default to False.

CHAPTER 9

Polls of taggers threads

class treetaggerwrapper.**TaggerPoll**(*workerscount=None*, *taggerscount=None*, ***kwargs*)

Keep a poll of TreeTaggers for processing with different threads.

This class is here for people preferring natural language processing over multithread programming… :-)

Each poll manage a set of threads, able to do parallel chunking, and a set of taggers, able to do (more real) parallel tagging. All taggers in the same poll are created for same processing (with same options).

`TaggerPoll` objects has same high level interface than `TreeTagger` ones with `_async` at end of methods names. Each of …_asynch method returns a `Job` object allowing to know if processing is finished, to wait for it, and to get the result.

If you want to **properly terminate** a `TaggerPoll`, you must call its `TaggerPoll.stop_poll()` method.

Note: Parallel processing via threads in Python within the same process is limited due to the global interpreter lock (Python's GIL). See *Polls of taggers process* for real parallel process.

Example of use

In this example no parameter is given to the poll, it auto-adapt to the count of CPU cores.

```
import treetaggerwrapper as ttpw
p = ttpw.TaggerPoll()

res = []
text = "This is Mr John's own house, it's very nice."
print("Creating jobs")
for i in range(10):
    print("     Job", i)
    res.append(p.tag_text_async(text))
print("Waiting for jobs to be completed")
for i, r in enumerate(res):
    print("     Job", i)
    r.wait_finished()
```

(continues on next page)

(continued from previous page)

```
    print(r.result)
p.stop_poll()
print("Finished")
```

Creation of a new TaggerPoll.

By default a `TaggerPoll` creates same count of threads and of TreeTagger objects than there are CPU cores on your computer.

Parameters

- **workerscount** (*int*) – number of worker threads to create.
- **taggerscount** (*int*) – number of TreeTaggers objects to create.
- **kwargs** – same parameters as `TreeTagger.__init__()`.

tag_text_async(*text*, *numlines=False*, *tagonly=False*, *prepronly=False*, *tagblanks=False*, *notagurl=False*, *notagemail=False*, *notagip=False*, *notagdns=False*, *nosgmlsplit=False*)

See `TreeTagger.tag_text()` method and `TaggerPoll` doc.

Returns a `Job` object about the async process.

Return type `Job`

tag_file_async(*infilepath*, *encoding='utf-8'*, *numlines=False*, *tagonly=False*, *prepronly=False*, *tagblanks=False*, *notagurl=False*, *notagemail=False*, *notagip=False*, *notagdns=False*, *nosgmlsplit=False*)

See `TreeTagger.tag_file()` method and `TaggerPoll` doc.

Returns a `Job` object about the async process.

Return type `Job`

tag_file_to_async(*infilepath*, *outfilepath*, *encoding='utf-8'*, *numlines=False*, *tagonly=False*, *prepronly=False*, *tagblanks=False*, *notagurl=False*, *notagemail=False*, *notagip=False*, *notagdns=False*, *nosgmlsplit=False*)

See `TreeTagger.tag_file_to()` method and `TaggerPoll` doc.

Returns a `Job` object about the async process.

Return type `Job`

stop_poll()

Properly stop a `TaggerPoll`.

Takes care of finishing waiting threads, and deleting TreeTagger objects (removing pipes connexions to treetagger process).

Once called, the `TaggerPoll` is no longer usable.

class treetaggerwrapper.**Job**(*poll*, *methname*, *kwargs*)

Asynchronous job to process a text with a Tagger.

These objects are automatically created for you and returned by `TaggerPoll` methods `TaggerPoll.tag_text_async()`, `TaggerPoll.tag_file_async()` and `TaggerPoll.tag_file_to_async()`.

You use them to know status of the asynchronous request, eventually wait for it to be finished, and get the final result.

Variables

- **`finished`** – Boolean indicator of job termination.
- **`result`** – Final job processing result — or exception.

`wait_finished``()`

Lock on the Job event signaling its termination.

CHAPTER 10

Extra functions

Some functions can be of interest, eventually for another project.

`treetaggerwrapper.`**`blank_to_space`**(*text*)

Replace blanks characters by real spaces.

May be good to prepare for regular expressions & Co based on whitespaces.

Parameters **`text`** (*`string`*) – the text to clean from blanks.

Returns List of parts in their apparition order.

Return type [string]

`treetaggerwrapper.`**`blank_to_tag`**(*text*)

Replace blanks characters by corresponding SGML tags in a text.

Parameters **`text`** (*`string`*) – the text to transform from blanks.

Returns List of texts and sgml tags where there was a blank.

Return type list.

`treetaggerwrapper.`**`enable_debugging_log`**()

Setup logging module output.

This setup a log file which register logs, and also dump logs to stdout. You can just copy/paste and adapt it to make logging write to your own log files.

`treetaggerwrapper.`**`get_param`**(*paramname*, *paramsdict*, *defaultvalue*)

Search for a working parameter value.

It is searched respectively in:

1. parameters given at *`TreeTagger`* construction.
2. environment variables.
3. configuration file, in `[CONFIG]` section.
4. default value.

`treetaggerwrapper.`**`is_sgml_tag`**`(`*text*`)`

Test if a text is - completly - a SGML tag.

Parameters **`text`** (*`string`*) – the text to test.

Returns True if it's an SGML tag.

Return type boolean

`treetaggerwrapper.`**`load_configuration`**`()`

Load configuration file for the TreeTagger wrapper.

This file is used mainly to store last automatically found directory of TreeTagger installation. It can also be used ot override some default working parameters of this script.

`treetaggerwrapper.`**`locate_treetagger`**`()`

Try to find treetagger directory in some standard places.

If a location is already available in treetaggerwrapper config file, then the function first check if it is still valid, and if yes simply return this location.

A treetagger directory (any variation of directory name with *tree* and *tagger*, containing `lib` and `bin` subdirectories) is search:

- In user home directories and its subdirectories.
- In MacOSX user own library frameworks.
- In system wide standard installation directories (depend on used platform).

The found location, if any, is stored into `treetagger_wrapper.cfg` file for later direct use (located in standard XDG config path).

If not found, the function returns None.

Returns directory conntaining TreeTagger installation, or None.

Return type str

`treetaggerwrapper.`**`main`**`(`**args*`)`

Test/command line usage code.

See command line usage help with:

```
python treetaggerwrapper.py --help
```

or:

```
python -m treetaggerwrapper --help
```

`treetaggerwrapper.`**`maketrans_unicode`**`(`*s1*, *s2*, *todel=''*`)`

Build translation table for use with unicode.translate().

Parameters

- **`s1`** (*`unicode`*) – string of characters to replace.
- **`s2`** (*`unicode`*) – string of replacement characters (same order as in s1).
- **`todel`** (*`unicode`*) – string of characters to remove.

Returns translation table with character code -> character code.

Return type dict

`treetaggerwrapper.`**`pipe_writer`**(*pipe*, *text*, *flushsequence*, *encoding*, *errors*)

Write a text to a pipe and manage pre-post data to ensure flushing.

For internal use.

If text is composed of str strings, they are written as-is (ie. assume ad-hoc encoding is providen by caller). If it is composed of unicode strings, then they are converted to the specified encoding.

Parameters

- **`pipe`** (*`Popen object (file-like with write and flush methods)`*) – the Popen pipe on what to write the text.
- **`text`** (*`string or list of strings`*) – the text to write.
- **`flushsequence`** (*`string (with n between tokens)`*) – lines of tokens to ensure flush by TreeTagger.
- **`encoding`** (*`str`*) – encoding of texts written on the pipe.
- **`errors`** (*`str`*) – how to manage encoding errors: strict/ignore/replace.

`treetaggerwrapper.`**`save_configuration`**()

Save configuration file for the TreeTagger wrapper.

`treetaggerwrapper.`**`split_sgml`**(*text*)

Split a text between SGML-tags and non-SGML-tags parts.

Parameters **`text`** (*`string`*) – the text to split.

Returns List of text/SgmlTag in their apparition order.

Return type list

CHAPTER 11

Polls of taggers process

Tests with *treetaggerwrapper.TaggerPoll* show limited benefit of multithreading processing, probably related to the large part of time spent in the preprocess chunking executed by Python code and dependant on the Python Global Interpreter Lock (GIL).

Another solution with Python standard packages is the `multiprocessing` module, which provides tools to dispatch computing between different process in place of threads, each process being independant with its own interpreter (so its own GIL).

The *treetaggerpoll* module and its class *TaggerProcessPoll* are for people preferring natural language processing over multiprocessing programming… :-)

A comparison using the following example, running on a Linux OS with 4 core Intel Xeon X5450 CPU, tested with 1 2 3 4 5 and 10 worker process, gives the result in table below — printed time is for the main process (which wait for its subprocess termination). This shows great usage of available CPU when using this module for chunking/tagging (we can see that having more worker process than CPU is not interesting — by default the class build as many worker process as you have CPUs):

Table 1: Workers count comparison

workers	printed time	real CPU time
1	228.49 sec	3m48.527s
2	87.88 sec	1m27.918s
3	61.12 sec	1m1.154s
4	53.86 sec	0m53.907s
5	50.68 sec	0m50.726s
10	56.45 sec	0m56.487s

11.1 Short example

This example is available in the source code repository, in `test/` subdirectory. Here you can see that main module must have its main code wrapped in a `if __name__ == '__main__':` condition (for correct Windows support).

It may take an optional parameter to select how many workers you want (by default as many workers as you have CPUs):

```python
import sys
import time

JOBSCOUNT = 10000

def start_test(n=None):
    start = time.time()
    import treetaggerpoll

    # Note: print() have been commented, you may uncomment them to see progress.
    p = treetaggerpoll.TaggerProcessPoll(workerscount=n, TAGLANG="en")
    res = []

    text = "This is Mr John's own house, it's very nice. " * 40

    print("Creating jobs")
    for i in range(JOBSCOUNT):
        # print("   Job", i)
        res.append(p.tag_text_async(text))

    print("Waiting for jobs to complete")
    for i, r in enumerate(res):
        # print("   Job", i)
        r.wait_finished()
        # print(str(r.result)[:50])
        res[i] = None   # Loose Job reference - free it.

    p.stop_poll()
    print("Finished after {:0.2f} seconds elapsed".format(time.time() - start))

if __name__ == '__main__':
    if len(sys.argv) >= 2:
        nproc = int(sys.argv[1])
    else:
        nproc = None
    start_test(nproc)
```

If you have a graphical CPU usage, you should see a high average load on each CPU.

Warning: Windows support

For Windows users, using `TaggerProcessPoll` have implications on your code, see multiprocessing docs, especially the Safe importing of main module part.

11.2 Main process poll classes

class treetaggerpoll.**TaggerProcessPoll**(*workerscount=None*, *keepjobs=True*, *wantresult=True*, *keeptagargs=True*, ***kwargs*)

Keep a poll of TreeTaggers process for processing with different threads.

Each poll manage a set of processes, able to do parallel chunking and tagging. All taggers in the same poll are created for same processing (with same options).

`TaggerProcessPoll` objects have same high level interface than `TreeTagger` ones with `_async` at end of methods names.

Each of `..._asynch` method returns a `ProcJob` object allowing to know if processing is finished, to wait for it, and to get the result.

If you want to **properly terminate** a `TaggerProcessPoll`, you must call its `TaggerProcessPoll.stop_poll()` method.

Creation of a new TaggerProcessPoll.

By default a `TaggerProcessPoll` creates same count of process than there are CPU cores on your computer .

Parameters

- **`workerscount`** (*`int`*) – number of worker process (and taggers) to create.
- **`keepjobs`** (*`bool`*) – poll keep references to Jobs to manage signal of their processing and store back processing results — default to True.
- **`wantresult`** (*`bool`*) – worker process must return processing result to be stored in the job — default to True.
- **`keeptagargs`** (*`bool`*) – must keep tagging arguments in `ProcJob` synchronization object — default to True.
- **`kwargs`** – same parameters as `treetaggerwrapper.TreeTagger.__init__()` for `TreeTagger` creation.

`tag_text_async`(*text*, *numlines=False*, *tagonly=False*, *prepronly=False*, *tagblanks=False*, *notagurl=False*, *notagemail=False*, *notagip=False*, *notagdns=False*, *nosgmlsplit=False*)

See `TreeTagger.tag_text()` method and `TaggerProcessPoll` doc.

Returns a `ProcJob` object about the async process.

Return type `ProcJob`

`tag_file_async`(*infilepath*, *encoding='utf-8'*, *numlines=False*, *tagonly=False*, *prepronly=False*, *tagblanks=False*, *notagurl=False*, *notagemail=False*, *notagip=False*, *notagdns=False*, *nosgmlsplit=False*)

See `TreeTagger.tag_file()` method and `TaggerProcessPoll` doc.

Returns a `ProcJob` object about the async process.

Return type `ProcJob`

`tag_file_to_async`(*infilepath*, *outfilepath*, *encoding='utf-8'*, *numlines=False*, *tagonly=False*, *prepronly=False*, *tagblanks=False*, *notagurl=False*, *notagemail=False*, *notagip=False*, *notagdns=False*, *nosgmlsplit=False*)

See `TreeTagger.tag_file_to()` method and `TaggerProcessPoll` doc.

Returns a `ProcJob` object about the async process.

Return type `ProcJob`

`stop_poll`()

Properly stop a `TaggerProcessPoll`.

Takes care of finishing waiting threads, and deleting TreeTagger objects (removing pipes connexions to treetagger process).

Once called, the `TaggerProcessPoll` is no longer usable.

class `treetaggerpoll.`**`ProcJob`**(*poll*, *methname*, *keepjobs*, *kwargs*)

Asynchronous job to process a text with a Tagger.

These objects are automatically created for you and returned by *`TaggerProcessPoll`* methods *`TaggerProcessPoll.tag_text_async()`*, *`TaggerProcessPoll.tag_file_async()`* and *`TaggerProcessPoll.tag_file_to_async()`*.

You use them to know status of the asynchronous request, eventually wait for it to be finished, and get the final result.

Note: If your *`TaggerProcessPoll`* has been created with `keepjobs` param set to False, you can't rely on the ProcJob object (neither finish state or result). And if you used `wantresult` param set to False, the final result can only be `"finished"` or an exception information string.

Variables

- **`finished`** – Boolean indicator of job termination.
- **`result`** – Final job processing result — or exception.

`wait_finished`()

Lock on the ProcJob event signaling its termination.

Python Module Index

Index