pyconll Documentation

Release 1.0.2

Matias Grioni

Dec 28, 2018

Contents

CHAPTER 1

pyconll

*Easily work with **CoNLL* files using the familiar syntax of **python**.*

The current version is 1.1.1. This version is fully functional, stable, tested, documented, and actively developed.

1.1 Links

- Homepage
- Documentation

1.1.1 Motivation

When working with the Universal Dependencies project, there are a dissapointing lack of low level APIs. There are many great tools, but few are general purpose enough. Grew is a great tool, but it is slightly limiting for some tasks (and extremely productive for others). Treex is similar to Grew in this regard. CL-CoNLLU is a good tool in this regard, but it is written in a language that many are not familiar with, Common Lisp. UDAPI might fit the bill with its python API, but the package itself is quite large and the documentation impossible to get through. Various more tools can be found on the Universal Dependencies website and all are very nice pieces of software, but most of them are lacking in this desired usage pattern. `pyconll` creates a thin API on top of raw CoNLL annotations that is simple and intuitive. This is an attempt at a small, minimal, and intuitive package in a popular language that can be used as building block in a complex system or the engine in small one off scripts.

Hopefully, individual researchers will find use in this project, and will use it as a building block for more popular tools. By using `pyconll`, researchers gain a standardized and feature rich base on which they can build larger projects and not worry about CoNLL annotation and output.

1.1.2 Code Snippet

```
import pyconll

UD_ENGLISH_TRAIN = './ud/train.conll'

train = pyconll.load_from_file(UD_ENGLISH_TRAIN)

for sentence in train:
    for token in sentence:
        # Do work here.
        if token.form == 'Spain':
            token.upos = 'PROPN'
```

More examples can be found in the `examples` folder.

1.1.3 Uses and Limitations

The usage of this package is to enable editing of CoNLL-U format annotations of sentences. Note that this does not include the actual text that is annotated. For this reason, word forms for Tokens are not editable and Sentence Tokens cannot be reassigned. Right now, this package seeks to allow for straight forward editing of annotation in the CoNLL-U format and does not include changing tokenization or creating completely new Sentences from scratch. If there is interest in this feature, it can be revisted for more evaluation.

1.1.4 Installation

As with most python packages, simply use `pip` to install from PyPi.

```
pip install pyconll
```

This package is designed for, and only tested with python 3.4 and above. Backporting to python 2.7 is not in future plans.

1.1.5 Documentation

The full API documentation can be found online at https://pyconll.readthedocs.io/. Examples can be found in the `examples` folder and also in the `tests` folder.

1.1.6 Contributing

If you would like to contribute to this project you know the drill. Either create an issue and wait for me to repond and fix it or ignore it, or create a pull request or both. When cloning this repo, please run `make hooks` and `pip install -r requirements.txt` to properly setup the repo. `make hooks` setups up the pre-push hook, which ensures the code you push is formatted according to the default YAPF style. `pip install -r requirements.txt` simply sets up the environment with dependencies like `yapf`, `twine`, `sphinx`, and so on.

README and CHANGELOG

When changing either of these files, please run `make docs` so that the `.rst` versions stay in sync. The main version is the markdown version.

Code Formatting

Code formatting is done automatically on push if githooks are setup properly. The code formatter is YAPF, and using this ensures that new code stays in the same style.

CHAPTER 2

CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

2.1 [1.1.1] - 2018-12-10

2.1.1 Fixed

- The `pyconll.tree` module was not properly included before in `setup.py`

2.2 [1.1.0] - 2018-11-11

2.2.1 Added

- `pylint` to build process
- `Conllable` abstract base class to mark CoNLL serializable components
- Tree data type construction of a sentence

2.2.2 Changed

- Linting patches suggested by `pylint`.
- Removed `_end_line_number` from `Sentence` constructor. This is an internal patch, as this parameter was not meant to be used by callers.
- New, improved, and clearer documentation
- Update of `requests` dependency due to security flaw

2.3 [1.0.1] - 2018-09-14

2.3.1 Changed

- Removed test packages from final shipped package.

2.4 [1.0] - 2018-09-13

2.4.1 Added

- There is now a FormatError to help make debugging easier if the internal data of a Token is put into an invalid state. This error will be seen on running `Token#conll`.
- Certain token fields with empty values, were not output when calling `Token#conll` and were instead ignored. This situation now causes a FormatError.
- Stricter parsing and validation of general CoNLL guidelines.

2.4.2 Fixed

- `DEPS` parsing was broken before and assumed that there was less information than is actually possible in the UD format. This means that now `deps` is a tuple with cardinality 4.

2.5 [0.3.1] - 2018-08-08

2.5.1 Fixed

- Fixed issue with submodules not being packaged in build

2.6 [0.3] - 2018-07-28

2.6.1 Added

- Ability to easily load CoNLL files from a network path (url)
- Some parsing validation. Before the error was not caught up front so the error could unexpectedly later show up.
- Sentence slicing had an issue before if either the start or end was omittted.
- More documentation and examples.
- Conll is now a `MutableSequence`, so it handles methods beyond its implementation as well as defined by python.

2.6.2 Fixed

- Some small bug fixes with parsing the token dicts.

2.7 [0.2.3] - 2018-07-23

2.7.1 Fixed

- Issues with documentation since docstrings were not in RST. Fixed by using napoleon sphinx extension

2.7.2 Added

- A little more docs
- More README info
- Better examples

2.8 [0.2.2] - 2018-07-18

2.8.1 Fixed

- Installation issues again with wheel when using `pip`.

2.9 [0.2.1] - 2018-07-18

2.9.1 Fixed

- Installation issues when using `pip`

2.10 [0.2] - 2018-07-16

2.10.1 Added

- More documentation
- Util package for convenient and common logic

2.11 [0.1.1] - 2018-07-15

2.11.1 Added

- Documentation which can be found here.
- Small documentation changes on methods.

2.12 [0.1] - 2018-07-04

2.12.1 Added

- Everything. This is the first release of this package. The most notable absence is documentation which will be coming in a near-future release.

CHAPTER 3

conllable

`Conllable` marks a class that can be output as a CoNLL formatted string. `Conllable` classes implement a `conll` method.

3.1 API

Holds the Conllable interface, which is a marker interface to show that a class is a Conll object, such as a treebank, sentence, or token, and therefore has a conll method.

class pyconll.conllable.**Conllable**

A Conllable mixin to indicate that the component can be converted into a CoNLL representation.

__metaclass__

alias of `abc.ABCMeta`

conll()

Provides a conll representation of the component.

Returns A string conll representation of the base component.

Raises `NotImplementedError` – If the child class does not implement the method.

CHAPTER 4

exception

Custom exceptions for pyconll. These errors are a `ParseError` and a `FormatError`.

A `ParseError` occurs when the source input to a CoNLL component is invalid, and a `FormatError` occurs when the internal state of the component is invalid, and the component cannot be output to a CoNLL string.

4.1 API

Holds custom pyconll errors. These errors include parsing errors when reading treebanks, and errors when outputting CoNLL objects.

exception pyconll.exception.**FormatError**
Error that results from trying to format a CoNLL object to a string.

exception pyconll.exception.**ParseError**
Error that results from an improper value into a parsing routine.

CHAPTER 5

load

This is the main module to interface with to load an entire CoNLL treebank resources. The module defines methods for loading a CoNLL treebank through a string, file, or network. There also exist methods that iterate over the CoNLL resource data rather than storing the large CoNLL object in memory, if so desired.

Note that the fully qualified name is `pyconll.load`, but these methods can also be accessed using the `pyconll` namespace.

5.1 Example

This example counts the number of times a token with a lemma of `linguistic` appeared in the treebank. If all the operations that will be done on the CoNLL file are readonly or are data aggregations, the `iter_from` alternatives are more efficient and recommended. These methods will return an iterator over the sentences in the CoNLL resource rather than storing the CoNLL object in memory, which can be convenient when dealing with large files that do not need be completely loaded.

```
import pyconll

example_treebank = '/home/myuser/englishdata.conll'
conll = pyconll.iter_from_file(example_treebank)

count = 0
for sentence in conll:
    for word in sentence:
        if word.lemma == 'linguistic':
            count += 1

print(count)
```

5.2 API

A wrapper around the Conll class that allow for easy loading of treebanks from multiple formats. This module also contains logic for iterating over treebank data without storing Conll objects in memory.

pyconll.load.**iter_from_file**(*filename*)
Iterate over a CoNLL-U file's sentences.

Parameters **filename** – The name of the file whose sentences should be iterated over.

Yields The sentences that make up the CoNLL-U file.

Raises

- IOError if there is an error opening the file.
- ParseError – If there is an error parsing the input into a Conll object.

pyconll.load.**iter_from_string**(*source*)
Iterate over a CoNLL-U string's sentences.

Use this method if you only need to iterate over the CoNLL-U file once and do not need to create or store the Conll object.

Parameters **source** – The CoNLL-U string.

Yields The sentences that make up the CoNLL-U file.

Raises ParseError – If there is an error parsing the input into a Conll object.

pyconll.load.**iter_from_url**(*url*)
Iterate over a CoNLL-U file that is pointed to by a given URL.

Parameters **url** – The URL that points to the CoNLL-U file.

Yields The sentences that make up the CoNLL-U file.

Raises

- requests.exceptions.RequestException – If the url was unable to be properly retrieved.
- ParseError – If there is an error parsing the input into a Conll object.

pyconll.load.**load_from_file**(*filename*)
Load a CoNLL-U file given the filename where it resides.

Parameters **filename** – The location of the file.

Returns A Conll object equivalent to the provided file.

Raises

- IOError – If there is an error opening the given filename.
- ParseError – If there is an error parsing the input into a Conll object.

pyconll.load.**load_from_string**(*source*)
Load CoNLL-U source in a string into a Conll object.

Parameters **source** – The CoNLL-U formatted string.

Returns A Conll object equivalent to the provided source.

Raises ParseError – If there is an error parsing the input into a Conll object.

pyconll.load.**load_from_url**(*url*)

Load a CoNLL-U file that is pointed to by a given URL.

Parameters **url** – The URL that points to the CoNLL-U file.

Returns A Conll object equivalent to the provided file.

Raises

- requests.exceptions.RequestException – If the url was unable to be properly retrieved and status was 4xx or 5xx.
- ParseError – If there is an error parsing the input into a Conll object.

CHAPTER 6

sentencetree

A `SentenceTree` is a thin wrapper around a `Sentence` that also provides a tree based representation of the sentence. The sentence for a `SentenceTree` can be retreived through the `sentence` property and the tree through the `tree` property.

This wrapper is very bare currently and only seeks to create the tree based representation, and does not provide additional logic. Please create a github issue if you would like to see functionality added in this area!

6.1 API

Create the SentenceTree type as a wrapper around a sentence that constructs a tree as well to traverse the sentence in a new way.

class pyconll.tree.sentencetree.**SentenceTree**(*sentence*)

A Tree wrapper around a sentence. This type will take in an existing serial sentence, and create a tree representation from it. This type holds both the sentence and the tree representation of the sentence. Note that an empty sentence input will have no data and no children.

__init__(*sentence*)

Creates a new SentenceTree given the sentence.

Parameters **sentence** – The sentence to wrap and construct a tree from.

conll()

Outputs the provided tree into CoNLL format.

Returns The CoNLL formatted string.

sentence

Provides the unwrapped sentence. This property is readonly.

Returns The unwrapped sentence.

tree

Provides the constructed tree. This property is readonly.

Returns The constructed tree.

CHAPTER 7

tree

`Tree` is a very basic immutable tree class. A `Tree` can have multiple children and has one parent. The parent of a tree is established when a `Tree` is added as the child of another `Tree`.

7.1 API

Defines a base immutable tree type. This type can then be used to create a TokenTree which maps a sentence. This type is meant to be limited in scope and use and not as a general tree builder module.

class pyconll.tree.tree.**Tree**(*data*, *children*)

A tree node. This is the base representation for a tree, which can have many children which are accessible via child index. The tree's structure is immutable, so the parent and children cannot be changed once created.

__getitem__(*key*)

Get specific children from the Tree. This can be an integer or slice.

Parameters **key** – The indexer for the item.

__init__(*data*, *children*)

Create a new tree with the desired properties.

Parameters

- **data** – The data to store on the tree.
- **children** – The children of this node. None if there are no children.

__iter__()

Provides an iterator over the children.

__len__()

Provides the number of direct children on the tree.

Returns The number of direct children on the tree.

children

Provides the children of the Tree. The property ensures it is readonly.

Returns The list of children nodes.

parent

Provides the parent of the Tree. The property ensures it is readonly.

Returns A pointer to the parent Tree reference.

CHAPTER 8

util

This module provides additional, common methods that build off of the API layer. This module simply adds logic, rather than extending the API. Right now this module is pretty sparse, but will be extended as needed.

8.1 API

A set of utilities for dealing with pyconll defined types. This is simply a collection of functions.

pyconll.util.**find_ngrams**(*conll*, *ngram*, *case_sensitive=True*)

Find the occurences of the ngram in the provided Conll collection.

This method returns every sentence along with the token position in the sentence that starts the ngram. The matching algorithm does not currently account for multiword tokens, so "don't" should be separated into "do" and "not" in the input.

Parameters

- **sentence** – The sentence in which to search for the ngram.
- **ngram** – The ngram to search for. A random access iterator.
- **case_sensitive** – Flag to indicate if the ngram search should be case sensitive.

Returns An iterator over the ngrams in the Conll object. The first element is the sentence and the second element is the numeric token index.

CHAPTER 9

conll

A collection of CoNLL annotated sentences. For creating new instances of this object, API callers should use the `pyconll.load` module to abstract over the resource type. The `Conll` object can be thought of as a simple wrapper around a list of sentences that can be serialized into a CoNLL format.

`Conll` is a subclass of `MutableSequence`, so `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__` are available free of charge, even though they are not defined below.

9.1 API

Defines the Conll type and the associated parsing and output logic.

class pyconll.unit.conll.**Conll**(*it*)

The abstraction for a CoNLL-U file. A CoNLL-U file is more or less just a collection of sentences in order. These sentences can be accessed by sentence id or by numeric index. Note that sentences must be separated by whitespace. CoNLL-U also specifies that the file must end in a new line but that requirement is relaxed here in parsing.

__contains__(*other*)

Check if the Conll object has this sentence.

Parameters **other** – The sentence to check for.

Returns True if this Sentence is exactly in the Conll object. False, otherwise.

__delitem__(*key*)

Delete the Sentence corresponding with the given key.

Parameters **key** – The info to get the Sentence to delete. Can be the integer position in the file, or a slice.

__getitem__(*key*)

Index a sentence by key value.

Parameters **key** – The key to index the sentence by. This key can either be a numeric key, or a slice.

Returns The corresponding sentence if the key is an int or the sentences if the key is a slice in the form of another Conll object.

Raises `TypeError` – If the key is not an integer or slice.

__init__(*it*)
Create a CoNLL-U file collection of sentences.

Parameters **it** – An iterator of the lines of the CoNLL-U file.

Raises `ParseError` – If there is an error constructing the sentences in the iterator.

__iter__()
Allows for iteration over every sentence in the CoNLL-U file.

Yields An iterator over the sentences in this Conll object.

__len__()
Returns the number of sentences in the CoNLL-U file.

Returns The size of the CoNLL-U file in sentences.

__setitem__(*key*, *sent*)
Set the given location to the Sentence.

Parameters **key** – The location in the Conll file to set to the given sentence. This only accepts integer keys and accepts negative indexing.

conll()
Output the Conll object to a CoNLL-U formatted string.

Returns The CoNLL-U object as a string. This string will end in a newline.

insert(*index*, *value*)
Insert the given sentence into the given location.

This function behaves in the same way as python lists insert.

Parameters

- **index** – The numeric index to insert the sentence into.
- **value** – The sentence to insert.

write(*writable*)
Write the Conll object to something that is writable.

For simply writing, this method is more efficient than calling conll then writing since no string of the entire Conll object is created. The final output will include a final newline.

Parameters **writable** – The writable object such as a file. Must have a write method.

CHAPTER 10

sentence

The `Sentence` module represents an entire CoNLL sentence, which is composed of two main parts: the comments and the tokens.

10.1 Comments

Comments are treated as key-value pairs, where the separating character between key and value is =. If there is no = present then then the comment is treated as a singleton, where the key is the comment string and the corresponding value is `None`. Read and write methods on this data can be found on methods prefixed with `meta_`.

For convenience, the id and text of a sentence can be accessed through member properties directly rather than through metadata methods. So `sentence.id`, rather than `sentence.meta_value('id')`. Since this API does not support changing a token's form, the `text` comment cannot be changed.

10.2 Document and Paragraph ID

The document and paragraph id of a sentence are automatically inferred from a CoNLL treebank given sentence comments. Reassigning ids must be done through comments on the sentence level, and there is no API for simplifying this reassignment.

10.3 Tokens

These are the meat of the sentence. Tokens can be accessed through their id defined in the CoNLL annotation as a string or as a numeric index. So the same indexing syntax understands, `sentence['5']`, `sentence['2-3']` and `sentence[2]`.

10.4 API

Defines the Sentence type and the associated parsing and output logic.

class pyconll.unit.sentence.**Sentence**(*source*, *_start_line_number=None*)
A sentence in a CoNLL-U file. A sentence consists of several components.

First, are comments. Each sentence must have two comments per UD v2 guidelines, which are sent_id and text. Comments are stored as a dict in the meta field. For singleton comments with no key-value structure, the value in the dict has a value of None.

Note the sent_id field is also assigned to the id property, and the text field is assigned to the text property for usability, and their importance as comments. The text property is read only along with the paragraph and document id. This is because the paragraph and document id are not defined per Sentence but across multiple sentences. Instead, these fields can be changed through changing the metadata of the Sentences.

Then comes the token annotations. Each sentence is made up of many token lines that provide annotation to the text provided. While a sentence usually means a collection of tokens, in this CoNLL-U sense, it is more useful to think of it as a collection of annotations with some associated metadata. Therefore the text of the sentence cannot be changed with this class, only the associated annotations can be changed.

__eq__(*other*)
Defines equality for a sentence.

Parameters **other** – The other Sentence to compare for equality against this one.

Returns True if the this Sentence and the other one are the same. Sentences are the same when their comments are the same and their tokens are the same. Line numbers are not including in the equality definition.

__getitem__(*key*)
Return the desired tokens from the Sentence.

Parameters **key** – The indicator for the tokens to return. Can either be an integer, a string, or a slice. For an integer, the numeric indexes of Tokens are used. For a string, the id of the Token is used. And for a slice the start and end must be the same data types, and can be both string and integer.

Returns If the key is a string then the appropriate Token. The key can also be a slice in which case a list of tokens is provided.

__init__(*source*, *_start_line_number=None*)
Construct a Sentence object from the provided CoNLL-U string.

Parameters

- **source** – The raw CoNLL-U string to parse. Comments must precede token lines.
- **_start_line_number** – The starting line of the sentence. For internal use.

Raises ParseError – If there is any token that was not valid.

__iter__()
Iterate through all the tokens in the Sentence including multiword tokens.

__len__()
Get the length of this sentence.

Returns The amount of tokens in this sentence. In the CoNLL-U sense, this includes both all the multiword tokens and their decompositions.

conll()
Convert the sentence to a CoNLL-U representation.

Returns A string representing the Sentence in CoNLL-U format.

doc_id
Get the document id associated with this Sentence. Read-only.

Returns The document id or None if no id is associated.

id
Get the sentence id.

Returns The sentence id. If there is none, then returns None.

meta_present(*key*)
Check if the key is present as a singleton or as a pair.

Parameters **key** – The value to check for in the comments.

Returns True if the key was provided as a singleton or as a key value pair. False otherwise.

meta_value(*key*)
Returns the value associated with the key in the metadata (comments).

Parameters **key** – The key whose value to look up.

Returns The value associated with the key as a string. If the key is a singleton then None is returned.

Raises `KeyError` – If the key is not present in the comments.

par_id
Get the paragraph id associated with this Sentence. Read-only.

Returns The paragraph id or None if no id is associated.

set_meta(*key*, *value=None*)
Set the metadata or comments associated with this Sentence.

Parameters

- **key** – The key for the comment.
- **value** – The value to associate with the key. If the comment is a singleton, this field can be ignored or set to None.

text
Get the continuous text for this sentence. Read-only.

Returns The continuous text of this sentence. If none is provided in comments, then None is returned.

CHAPTER 11

token

The Token module represents a CoNLL token annotation. In a CoNLL file, this corresponds to a non-empty, non-comment line. `Token` members correspond directly with the Universal Dependencies CoNLL definition and all values are stored as strings. This means ids are strings as well. These fields are: `id`, `form`, `lemma`, `upos`, `xpos`, `feats`, `head`, `deprel`, `deps`, `misc`

11.1 Fields

All fields are strings except for `feats`, `deps`, and `misc`, which are `dicts`. Each of these fields has specific semantics per the UDv2 guidelines.

Since all of these fields are `dicts`, modifying non existent keys will result in a `KeyError`. This means that new values must be added as in a normal `dict`. For `set` based `dicts`, `feats` and specific fields of `misc`, the new key must be assigned to an empty `set` to start. More details on this below.

11.1.1 feats

`feats` is a key value mapping from `str` to `set`. Note that any keys with empty `sets` will throw an error, as all keys must have at least one feature.

11.1.2 deps

`deps` is a key value mapping from `str` to `tuple` of cardinality 4. Most Universal Dependencies treebanks, only use 2 of these 4 dimensions: the token index and the relation. See the Universal Dependencies guideline for more information on these 4 components.When adding new `deps`, the values must also be tuples of cardinality 4. Note that `deps` parsing is broken before version 1.0.

11.1.3 misc

Lastly, for `misc`, the documentation only specifies that the values are separated by a '|'. So not all components have to have a value. So, the values on `misc` are either `None` for entries with no '=', or `set` of `str`. A key with a value of `None` is output as a singleton.

11.1.4 Example

Below is an example of adding a new feature to a token, where the key must first be initialized:

```
token.feats['NewFeature'] = set(('No', ))
```

or alternatively as:

```
token.feats['NewFeature'] = set()
token.feats['NewFeature'].add('No')
```

11.2 API

Defines the Token type and the associated parsing and output logic.

class pyconll.unit.token.**Token**(*source*, *empty=True*, *_line_number=None*)

A token in a CoNLL-U file. This consists of 10 columns, each separated by a single tab character and ending in an LF ('n') line break. Each of the 10 column values corresponds to a specific component of the token, such as id, word form, lemma, etc.

This class does not do any formatting validation on input or output. This means that invalid input may be properly processed and then output. Or that client changes to the token may result in invalid data that can then be output. Properly formatted CoNLL-U will always work on input and as long as all basic units are strings output will work as expected. The result may just not be proper CoNLL-U.

Also note that the word form for a token is immutable. This is because CoNLL-U is inherently interested in annotation schemes and not storing sentences.

__eq__(*other*)

Test if this Token is equal to other.

Parameters **other** – The other token to compare against.

Returns True if the this Token and the other are the same. Two tokens are considered the same when all columns are the same.

__init__(*source*, *empty=True*, *_line_number=None*)

Construct the token from the given source.

A Token line must end in an an LF line break according to the specification. However, this method will accept a line with or without this ending line break.

Further, a '_' that appears in the form and lemma is ambiguous and can either refer to an empty value or an actual underscore. So the flag empty_form allows for control over this if it is known from outside information. If, the token is a multiword token, all fields except for form should be empty.

Note that no validation is done on input. Valid input will be processed properly, but there is no guarantee as to invalid input that does not follow the CoNLL-U specifications.

Parameters

- **line** – The line that represents the Token in CoNLL-U format.

- **empty** – A flag to signify if the word form and lemma can be assumed to be empty and not the token signifying empty. Only if both the form and lemma are both the same token as empty and there is no empty assumption, will they not be assigned to None.
- **_line_number** – The line number for this Token in a CoNLL-U file. For internal use mostly.

Raises `ParseError` – If the provided source is not composed of 10 tab separated columns.

conll()
Convert Token to the CoNLL-U representation.

Note that this does not include a newline at the end.

Returns A string representing the token as a line in a CoNLL-U file.

form
Provide the word form of this Token. This property makes it readonly.

Returns The Token wordform.

is_multiword()
Checks if this token is a multiword token.

Returns True if this token is a multiword token, and False otherwise.

This is the homepage for `pyconll` documentation. Here you can find module interfaces, changelogs, and example code. Simply look above to the table of contents for more info.

If you are looking for example code, please see the `examples` directory on github.

Python Module Index

p

Index

Symbols

C

D

F

I

L

M

P

S

T

W