# pyconll Documentation

*Release 0.1*

Matias Grioni

Jul 20, 2018

# Contents

CHAPTER 1

# pyconll

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

The current version is 0.2.1. This version is fully functional, stable, tested and documented.

## 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.

## 1.2 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.

## 1.3 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.

## 1.4 Documentation

Documentation is essential to learning how to use any library. The full API documentation can be found online at https://pyconll.readthedocs.io/en/latest/. Examples are also important and a growing number of examples can be found in the `examples` folder.

## 1.5 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.

### 1.5.1 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.

### 1.5.2 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 [0.2.1] - 2018-07-18

### 2.1.1 Fixed

- Installation issues when using `pip`

## 2.2 [0.2] - 2018-07-16

### 2.2.1 Added

- More documentation
- Util package for convenient and common logic

## 2.3 [0.1.1] - 2018-07-15

### 2.3.1 Added

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

## 2.4 [0.1] - 2018-07-04

### 2.4.1 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

# load

This is the main module you should interface with if wanting to load an entire CoNLL file, rather than individual sentences which should be less common. The API allows for loading CoNLL data from a string or from a file, and allows for iteration over the data, rather than storing a 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.

## 3.1 Example

This example counts the number of times a token with a lemma of `linguistic` appeared in the treebank. Note that if all the operations that will be done on the CoNLL file are readonly, consider using the `iter_from` alternatives. These methods will return an iterator over each sentence in the CoNLL file rather than storing an entire CoNLL object in memory, which can be convenient when dealing with large files that do not need to persist.

```python
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)
```

## 3.2 API

`pyconll.load.`**`iter_from_file`**(*filename*)

Iterate over a CoNLL-U file's sentences.

Args: filename: The name of the file whose sentences should be iterated over.

Returns: An iterator that yields consecutive sentences.

`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.

Args: source: The CoNLL-U string.

Returns: An iterator that yields consecutive sentences.

`pyconll.load.`**`load_from_file`**(*filename*)

Load a CoNLL-U file given the filename where it resides.

Args: filename: The location of the file.

Returns: A Conll object equivalent to the provided file.

`pyconll.load.`**`load_from_string`**(*source*)

Load CoNLL-U source in a string into a Conll object.

Args: source: The CoNLL-U formatted string.

Returns: A Conll object equivalent to the provided source.

# CHAPTER 4

# util

This is module that provides some useful functionality on top of pyconll. This adds logic on top of the API layer rather than extending it. Right now this module is pretty sparse, but it can be easiy extended as demand arises.

## 4.1 API

`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.

Args: 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 5

# conll

A collection of CoNLL annotated sentences. This collection should rarely be created by API callers, that is what the `pyconll.load` module is for which allows for easy APIs to load CoNLL files from a string or file (no network yet). The Conll object can be thought of as a simple list of sentences. There is very little more of a wrapper around this.

## 5.1 API

**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.

**`append`**(*sent*)

Add the given sentence to the end of this Conll object.

Args: sent: The Sentence object to add.

**`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*, *sent*)

Insert the given sentence into the given location.

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

Args: index: The numeric index to insert the sentence into. sent: 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.

Args: writable: The writable object such as a file. Must have a write method.

# CHAPTER 6

# sentence

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

## 6.1 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 and the corresponding value is `None`. To access and write to these values look for values related to meta (the meta data of the sentence).

Some things to keep in mind is that the id and text of a sentence can be accessed through member properties directly rather than through method APIs. So `sentence.id`, rather than `sentence.meta_value('id')`. Note that since this API does not support changing the forms of tokens, and focuses on the annotation of tokens, the text value cannot be changed of a sentence, but all other meta values can be.

## 6.2 Document and Paragraph ID

Document and paragraph id of a sentence are automatically inferred from a CoNLL treebank given the comments on each sentence. Note that if you wish to reassign these ids, it will have to be at the sentence level, there is no simplifying API to allow for easier mass assignment of this.

## 6.3 Tokens

These are the meat of the sentence. Some things to note for tokens are that they can be accessed either through id as defined in the CoNLL data as a string or as numeric index. The string id indexing allows for multitoken and null nodes to be included easily. So the same indexing syntax understands both, `sentence['2-3']` and `sentence[2]`.

## 6.4 API

**class** `pyconll.unit.sentence.`**`Sentence`**(*source*, *_start_line_number=None*, *_end_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.

**`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.

Args: 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).

Args: key: The key whose value to look up.

Returns: The value associated with the key as a string. If the key is not present then a KeyError is thrown, and if the key is a singleton then None is returned.

**`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.

Args: 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 7

# token

The Token module represents a single token (multiword or otherwise) in a CoNLL-U file. In text, this corresponds to one non-empty, non-comment line. Token has several members that correspond with the columns of the lines. All values are stored as strings. So ids are strings and not numeric. These fields are listed below and coresspond exactly with those found in the Universal Dependencenies project:

id form lemma upos xpos feats head deprel deps misc

## 7.1 Fields

Currently, all fields are strings except for `feats`, `deps`, and `misc`, which are `dicts`. There are specific semantics for each of these according to the UDv2 guidelines. The current approach is for these fields to be `dicts` as described below rather than providing an extra interface for these fields.

### 7.1.1 feats

`feats` is a dictionary of attribute value pairs, where there can be multiple values. So the values for `feats` is a `set` when parsed. When assigning and changing the values in `feats` ensure that the values are `sets`. Note that any keys with empty `sets` will not be output.

### 7.1.2 deps

`deps` is also a dictionary of attribute value pairs, but there is only one value, so the values are strings.

### 7.1.3 misc

Lastly, for `misc`, the documentation only specifies that the values are separated by a '|'. So for this reason, the value for `misc` is either `None` for entries with no '=', and an attribute value pair otherwise with the value being a `string` or a `set`.

## 7.2 API

**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.

Args: 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.

Args: 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: ValueError if the provided source is not composed of 10 tab separated columns.

**__weakref__**

list of weak references to the object (if defined)

**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 of the `pyconll` documentation. Here you can find most information you need to about module interfaces, changes in previous versions, and example code. Simply look to the table of contents above for more info.

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

# Python Module Index

# Index