pyconll Documentation

Release 3.0.0

Matias Grioni

Feb 24, 2021

CONTENTS

Welcome to the `pyconll` documentation homepage.

pyconll is designed as a flexible wrapper around the CoNLL-U format, to allow for easy loading and manipulating of dependency annotations. See an example of pyconll's syntax below.

```
import pyconll

# Load from disk into memory and iterate over the corpus, printing
# sentence ids, and capturing unique verbs
verbs = set()
corpus = pyconll.load_from_file('ud-english-train.conllu')
for sentence in corpus:
   print(sentence.id)
   for token in sentence:
      if token.upos == 'VERB':
         verbs.add(token.lemma)

# Use the iterate version over a larger corpus to save memory
huge_corpus_iter = pyconll.iter_from_file('annotated_shakespeare.conllu')
for sentence in huge_corpus_iter:
   print(sentence.id)
```

Those new to the project should visit the Getting Started page which goes through an end-to-end example using pyconll. For loading a file visit the load page. For API usage, confer with the conll, sentence, and token module pages which contain documentation for the base data types. Module documentation, guidance pages, and more are listed below in the table of contents.

For more information, the github project page has examples, tests, and source code.

CHAPTER

ONE

GETTING STARTED

1.1 Overview

`pyconll` is a low level wrapper around the CoNLL-U format. This document explains how to quickly get started loading and manipulating CoNLL-U files within `pyconll`, and will go through a typical end-to-end scenario.

To install the library, run `pip install pyconll` from your python enlistment.

1.2 Loading CoNLL-U

To start, a CoNLL-U resource must be loaded, and `pyconll` can load from files, urls, and strings. Specific API information can be found in the load module documentation. Below is a typical example of loading a file on the local computer.

```
import pyconll

my_conll_file_location = './ud/train.conllu'
train = pyconll.load_from_file(my_conll_file_location)
```

Loading methods usually return a `Conll` object, but some methods return an iterator over `Sentences` and do not load the entire `Conll` object into memory at once. The `Conll` object satisfies the MutableSequence contract in python, which means it functions nearly the same as a *list*.

1.3 Traversing CoNLL-U

After loading a CoNLL-U file, we can traverse the Conll structure. Conll objects wrap Sentences and Sentences wrap Tokens. Here is what traversal normally looks like.

```
for sentence in train:
    for token in sentence:
        # Do work within loops
        pass
```

Statistics such as lemmas for a certain closed class POS or number of non-projective punctuation dependencies can be computed through these loops. As an abstract example, we have defined some predicate, `sentence_pred`, and some transformation of noun tokens, `noun_token_transformation`, and we wish to transform all nouns in sentences that match our predicate, we can write the following.

```
for sentence in train:
    if sentence_pred(sentence):
        for token in sentence:
            if token.pos == 'NOUN':
                noun_token_transformation(token)
```

Note that most objects in `pyconll` are mutable, except for a select few fields, so changes on the `Token` object remain with the `Sentence` and can be output back into CoNLL format when processing is complete.

1.4 Outputting CoNLL-U

Once you are done working with a `Conll` object, you may need to output your results. The object can be serialized back into the CoNLL-U format, through the `conll` method. `Conll`, `Sentence`, and `Token` objects are all Conllable which means they have a corresponding `conll` method which serializes the objects into the appropriate string representation.

A more efficient way of outputting an entire Conll file would be to use the `write` method, which prevents creating the entire Conll file string in memory. When creating the file to write to, remember that, CoNLL-U is UTF-8 encoded.

1.5 Complete example

Putting together all the above elements, a complete example from loading, to transformation, to output looks as follows.

```
import pyconll

# Load file
my_conll_file_location = './ud/train.conllu'
train = pyconll.load_from_file(my_conll_file_location)

# Process and transform
for sentence in train:
    if sentence_pred(sentence):
        for token in sentence:
            if token.pos == 'NOUN':
                noun_token_transformation(token)

# Output changes. This writes directly to file, an alternative is to use
# train.conll() which will return the entire output string at once.
with open('output.conllu', 'w', encoding='utf-8') as f:
    train.write(f)
```

1.6 Conclusion

`pyconll` allows for easy CoNLL-U loading, traversal, and serialization. Developers can define their own transformation or analysis of the loaded CoNLL-U data, and pyconll handles all the parsing and serialization logic. There are still some parts of the library that are not covered here such as the `Tree` data structure, loading files from resources other than files, and error handling, but the information on this page will get developers through the most important use cases.

CHAPTER

TWO

LOAD

This module defines the main interface to load CoNLL treebank resources. CoNLL treebanks can be loaded through a string or a file (or technically anything that can function as a string iterator). CoNLL resources can be loaded and held in memory, or simply iterated through a sentence at a time which is useful for handling very large files.

The fully qualified name of the module is `pyconll.load`, but these methods are imported at the `pyconll` namespace level. This module provides the wrappers for loading from a string or file, but if another string iterator is available, for example, a network resource, this can be passed directly to the Conll constructor as well.

2.1 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 memory efficient alternative as well. 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. This example uses the `load_from_file` method for illustration purposes.

```
import pyconll

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

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

print(count)
```

2.2 API

A wrapper around the Conll class to easily load treebanks from multiple formats. This module can also load resources by iterating over treebank data without storing Conll objects in memory. This module is the main entrance to pyconll's functionalities.

`pyconll.load.`**`iter_from_file`**(*filename: str*) → Iterator[*pyconll.unit.sentence.Sentence*]

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: str*) → Iterator[*pyconll.unit.sentence.Sentence*]

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.**load_from_file**(*filename: str*) → *pyconll.unit.conll.Conll*

Load a CoNLL-U file given its location.

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: str*) → *pyconll.unit.conll.Conll*

Load the 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.

CHAPTER

THREE

TOKEN

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

3.1 Fields

All fields are optional strings except for `feats`, `deps`, and `misc`, which are `dicts`. As optional strings, they can either be None, or a string value. Fields which are dictionaries have specific semantics per the UDv2 guidelines. Since these fields are `dicts` this means modifying them uses python's natural syntax for dictionaries.

3.1.1 feats

`feats` is a key-value mapping from `str` to `set`. An example entry would be key `Gender` with value `set((Feminine,))`. More features could be added to an existing key by adding to its set, or a new feature could be added by adding to the dictionary. All features must have at least one value, so any keys with empty sets will throw an error on serialization back to text.

3.1.2 deps

`deps` is a key-value mapping from `str` to `tuple` of cardinality 4. This field represents enhanced dependencies. The key is the index of the token head, and the tuple elements define the enhanced dependency. 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.

3.1.3 misc

For `misc`, the documentation only specifies that values be separated by a '|', so not all keys have to have a value. So, values on `misc` are either `None`, or a `set` of `str`. A key with a value of `None` is output as a singleton, with no separating '='. A key with a corresponding `set` value will be handled like `feats`.

3.1.4 Examples

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')
```

On the miscellaneous column, adding a singleton field is done with the following line:

```
token.misc['SingletonFeature'] = None
```

3.2 API

Defines the Token type and parsing and output logic. A Token is the based unit in CoNLL-U and so the data and parsing in this module is central to the CoNLL-U format.

class pyconll.unit.token.**Token**(*source: str*, *empty: bool = False*)

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.

__init__(*source: str*, *empty: bool = False*) → None

Construct a Token from the given source line.

A Token line ends in an an LF line break according to the CoNLL-U specification. However, this method accepts a line with or without the LF line break.

On parsing, a '_' in the form and lemma is ambiguous and either refers to an empty value or to an actual underscore. The empty parameter flag controls how this situation should be handled.

This method also guarantees properly processing valid input, but invalid input may not be parsed properly. Some inputs that do not follow the CoNLL-U specification may still be parsed properly and as expected. So proper parsing is not an indication of validity.

Parameters

- **line** – The line that represents the Token in CoNLL-U format.
- **empty** – A flag to control if the word form and lemma can be assumed to be empty and not the token signifying empty. If both the form and lemma are underscores and empty is set to False (there is no empty assumption), then the form and lemma will be underscores rather than None.

Raises ***ParseError*** – On various parsing errors, such as not enough columns or improper column values.

conll() → str
Convert this Token to its CoNLL-U representation.

A Token's CoNLL-U representation is a line. Note that this method does not include a newline at the end.

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

property form
Provide the word form of this Token. This property is read only.

Returns The Token form.

is_multiword() → bool
Checks if this Token is a multiword token.

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

CHAPTER

FOUR

SENTENCE

The `Sentence` module represents an entire CoNLL sentence, which is composed of comments and tokens.

4.1 Comments

Comments are treated as key-value pairs, separated by the = character. A singleton comment has no = present. In this situation the key is the comment string, and the value is `None`. Methods for reading and writing cmoments on Sentences are prefixed with `meta_`, and are found below.

For convenience, the id and text comments are accessible through member properties on the Sentence in addition to metadata methods. So `sentence.id` and `sentence.meta_value('id')` are equivalent but the former is more concise and readable. Since this API does not support changing a token's form, the `text` comment cannot be changed. Text translations or transliterations can still be added just like any other comment.

4.1.1 Document and Paragraph ID

In previous versions of pyconll, the document and paragraph id of a Sentence were extracted similar to text and id information. This causes strange results and semantics when adding Sentences to a Conll object since the added sentence may have a `newpar` or `newdoc` comment which affects all subsequent Sentence ids. For simplicity's sake, this information is now only directly available as normal metadata information.

4.2 Tokens

This is the heart of the sentence. Tokens can be indexed on Sentences through their id value, as a string, or as a numeric index. So all of the following calls are valid, `sentence['5']`, `sentence['2-3']`, `sentence['2.1']`, and `sentence[2]`. Note that `sentence[x]` and `sentence[str(x)]` are not interchangeable. These calls are both valid but have different meanings.

4.3 API

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

class pyconll.unit.sentence.**Sentence**(*source: str*)
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.

__getitem__(*key: str*) → *pyconll.unit.token.Token*
__getitem__(*key: int*) → *pyconll.unit.token.Token*
__getitem__(*key: slice*) → Sequence[*pyconll.unit.token.Token*]
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 sequence of tokens is provided.

__init__(*source: str*) → 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.

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

__iter__() → Iterator[*pyconll.unit.token.Token*]
Iterate through all the tokens in the Sentence including multiword tokens.

__len__() → int
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() → str
Convert the sentence to a CoNLL-U representation.

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

property id
Get the sentence id.

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

meta_present(*key: str*) → bool
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: str*) → Optional[str]
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.

remove_meta(*key: str*) → None
Remove a metadata element associated with the Sentence.

Parameters **key** – The name of the metadata / comment.

Raises

- **KeyError** – If the key is not present in the Sentence metadata.
- **ValueError** – If the text key is provided, regardless of presence.

set_meta(*key: str*, *value: Optional[str] = None*) → None
Set or add 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.

property 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.

to_tree() → *pyconll.tree.tree.Tree*[*pyconll.unit.token.Token*]
Creates a Tree data structure from the current sentence.

An empty sentence will cannot be converted into a Tree and will throw an exception. The children for a node in the tree are ordered as they appear in the sentence. So the earliest child of a token appears first in the token's children in the tree.

Each Tree node has a data member that references the actual Token represented by the node. Multiword tokens are not included in the tree since they are more like virtual Tokens and do not participate in any dependency relationships or carry much value in dependency relations.

Returns A constructed Tree that represents the dependency graph of the sentence.

Raises **ValueError** – If the sentence can not be made into a tree because a token has an empty head value or if there is no root token.

CHAPTER

FIVE

CONLL

Note: For loading new `Conll` objects from a file or string, prefer the load module which provides the main entry points for parsing CoNLL.

This module represents a CoNLL file, i.e. a collection of CoNLL annotated sentences. Like other collections in python, `Conll` objects can be indexed, sliced, iterated, etc (specifically it implements the MutableSequence contract). `Conll` objects are Conllable, so then can be converted into a CoNLL string or they can be written to file directly with the `write` method.

5.1 API

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

class pyconll.unit.conll.**Conll**(*it: Iterable[str]*)

The abstraction for a CoNLL-U file. A CoNLL-U file is more or less just a collection of sentences in order. These sentences are accessed 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: object*) → bool

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: Union[int, slice]*) → None

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: int*) → *pyconll.unit.sentence.Sentence*
__getitem__(*key: slice*) → *Conll*

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: Iterable[str]*) → None
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__ () → Iterator[*pyconll.unit.sentence.Sentence*]
Allows for iteration over every sentence in the CoNLL-U file.

Yields An iterator over the sentences in this Conll object.

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

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

__setitem__ (*key: int*, *sent:* pyconll.unit.sentence.Sentence) → None
__setitem__ (*key: slice*, *sents: Iterable[*pyconll.unit.sentence.Sentence*]*) → None
Set the given location to the Sentence.

Parameters

- **key** – The location in the Conll file to set to the given sentence. This accepts integer or slice keys and accepts negative indexing.
- **item** – The item to insert. This can be an individual sentence, or another Conll object.

conll () → str
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: int*, *value:* pyconll.unit.sentence.Sentence) → None
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: Any*) → None
Write the Conll object to something that is writable.

For file writing, this method is more efficient than calling conll then writing since no string of the entire Conll object is created. The output includes a final newline as detailed in the CoNLL-U specification.

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

CHAPTER

SIX

TREE

`Tree` is a very basic immutable tree class. A `Tree` can have multiple children and has one parent. The parent and child of a tree are established when a `Tree` is created. Accessing the data on a `Tree` can be done through the `data` member. A `Tree`'s direct children can be iterated with a for loop. A `Tree` is created through the `TreeBuilder` module which is an internal module in pyconll and keeps `Tree` immutable on the public interface. A `Tree` is provided as the result of `to_tree` on a `Sentence` object.

6.1 API

A general immutable tree module. This module is used when parsing a serial sentence into a Tree structure.

class pyconll.tree.tree.**Tree**(*data: T*)

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 data, parent, children cannot be changed once created.

As is this class is useless, and must be created with the TreeBuilder module which is a sort of friend class of Tree to maintain its immutable public contract.

__getitem__(*key: int*) → *Tree*[T]
__getitem__(*key: slice*) → List[*Tree*[T]]

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

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

__init__(*data: T*) → None

Create a tree holding the value. Create a larger Tree, with TreeBuilder.

Parameters **data** – The data to put with the Tree node.

__iter__() → Iterator[*pyconll.tree.tree.Tree*[T]]

Provides an iterator over the children.

__len__() → int

Provides the number of direct children on the tree.

Returns The number of direct children on the tree.

property data

The data on the tree node. The property ensures it is readonly.

Returns The data stored on the Tree.

property parent

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

Returns A pointer to the parent Tree reference. None if there is no parent.

CHAPTER

SEVEN

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.

7.1 API

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

pyconll.util.**find_ngrams**(*conll: Iterable[*pyconll.unit.sentence.Sentence*], ngram: Sequence[str], case_sensitive: bool = True*) → Iterator[Tuple[*pyconll.unit.sentence.Sentence*, int, List[*pyconll.unit.token.Token*]]]

Find the occurrences 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

- **conll** – The corpus in which to search for the ngram across the sentences.
- **ngram** – The ngram to search for. An iterator of the lemmas.
- **case_sensitive** – Flag to indicate if the ngram search should be case sensitive. The case insensitive comparison currently is locale insensitive lowercase comparison.

Returns An iterator of tuples over the ngrams in the Conll object. The first element is the sentence, the second element is the numeric token index, and the last element is the actual list of tokens references from the sentence. This list does not include any multiword token that were skipped over.

pyconll.util.**find_nonprojective_deps**(*sentence:* pyconll.unit.sentence.Sentence) → List[Tuple[*pyconll.unit.token.Token*, *pyconll.unit.token.Token*]]

Find the nonprojective dependency pairs in the provided sentence.

Dependencies are provided as a list of ordered pairs. Each ordered pair represents a non-projective dependency pair. Each element in the ordered pair is a token, that makes a dependency with its governor. So each token is the base of its dependency, and the two tokens' dependencies cross in a non projective way.

Parameters **sentence** – The sentence to check for nonprojective dependency pairs.

Returns An list of pairs which represent the children of a nonprojective dependency pair.

CHAPTER

EIGHT

CONLLABLE

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

This is an abstract base class, and so no pure `Conllable` instance can be created. Instead this serves as an interface, marking types which have an expectation of supporting serialization. This consists of classes such as `Conll`, `Sentence`, and `Token`, which are unified by this interface.

8.1 API

The Conllable interface is a marker interface to show that a class is in the Conll object domain, such as a treebank (Conll in this library), 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.

abstract conll() → str

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

NINE

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.

9.1 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

TEN

PYCONLL

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

10.1 Links

- Homepage
- Documentation

10.1.1 Installation

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

```
pip install pyconll
```

`pyconll` is also available as a conda package on the `pyconll` channel. Only packages 2.2.0 and newer are available on conda at the moment.

```
conda install -c pyconll pyconll
```

pyconll supports Python 3.6 and greater, starting in version 3.0.0. In general pyconll will focus development efforts on officially supported python versions. Python 3.5 reached end of support in October 2020.

10.1.2 Use

This tool is intended to be a **minimal**, **low level**, **expressive** and **pragmatic** library in a widely used programming language. pyconll creates a thin API on top of raw CoNLL annotations that is simple and intuitive.

It offers the following features: * Regular CI testing and validation against all UD v2.x versions. * A strong domain model that includes CoNLL sources, Sentences, Tokens, Trees, etc. * A typed API for better development experience and better semantics. * A focus on usability and simplicity in design (no dependencies) * Performance optimizations for a smooth development workflow no matter the dataset size (performs about 25%-35% faster than other comparable packages)

See the following code example to understand the basics of the API.

```
# This snippet finds sentences where a token marked with part of speech 'AUX' are
# governed by a NOUN. For example, in French this is a less common construction
# and we may want to validate these examples because we have previously found some
# problematic examples of this construction.
import pyconll
```

(continues on next page)

(continued from previous page)

```
train = pyconll.load_from_file('./ud/train.conllu')

review_sentences = []

# Conll objects are iterable over their sentences, and sentences are iterable
# over their tokens. Sentences also de/serialize comment information.
for sentence in train:
    for token in sentence:

        # Tokens have attributes such as upos, head, id, deprel, etc, and sentences
        # can be indexed by a token's id. We must check that the token is not the
        # root token, whose id, '0', cannot be looked up.
        if token.upos == 'AUX' and (token.head != '0' and sentence[token.head].upos␣
↪== 'NOUN'):
            review_sentences.append(sentence)

print('Review the following sentences:')
for sent in review_sentences:
    print(sent.id)
```

A full definition of the API can be found in the documentation or use the quick start guide for a focused introduction.

10.1.3 Uses and Limitations

This package edits CoNLL-U annotations. This does not include the annotated text itself. Word forms on Tokens are not editable and Sentence Tokens cannot be reassigned or reordered. `pyconll` focuses on editing CoNLL-U annotation rather than creating it or changing the underlying text that is annotated. If there is interest in this functionality area, please create a GitHub issue for more visibility.

This package also is only validated against the CoNLL-U format. The CoNLL and CoNLL-X format are not supported, but are very similar. I originally intended to support these formats as well, but their format is not as well defined as CoNLL-U so they are not included. Please create an issue for visibility if this feature interests you.

Lastly, linguistic data can often be very large and this package attempts to keep that in mind. pyconll provides methods for creating in memory conll objects along with an iterate only version in case a corpus is too large to store in memory (the size of the memory structure is several times larger than the actual corpus file). The iterate only version can parse upwards of 100,000 words per second on a 16gb ram machine, so for most datasets to be used on a local dev machine, this package will perform well. The 2.2.0 release also improves parse time and memory footprint by about 25%!

10.1.4 Contributing

Contributions to this project are welcome and encouraged! If you are unsure how to contribute, here is a guide from Github explaining the basic workflow. After cloning this repo, please run `pip install -r requirements.txt` to properly setup locally. Some of these tools like yapf, pylint, and mypy do not have to be run locally, but CI builds will fail without their successful running. Some other release dependencies like twine and sphinx are also installed.

For packaging new versions, use setuptools version 24.2.0 or greater for creating the appropriate packaging that recognizes the `python_requires` metadata. Final packaging and release is now done with Github actions so this is less of a concern.

README and CHANGELOG

When changing either of these files, please change the Markdown version and run `make gendocs` so that the other versions stay in sync.

Release Checklist

Below enumerates the general release process explicitly. This section is for internal use and most people do not have to worry about this. First note, that the dev branch is always a direct extension of master with the latest changes since the last release. That is, it is essentially a staging release branch.

- Change the version in `pyconll/_version.py` appropriately.
- Merge dev into master **locally**. Github does not offer a fast forward merge and explicitly uses –no-ff. So to keep the linear nature of changes, merge locally to fast forward. This is assuming that the dev branch looks good on CI tests which do not automatically run in this situation.
- Push the master branch. This should start some CI tests specifically for master. After validating these results, create a tag corresponding to the next version number and push the tag.
- Create a new release from this tag from the Releases page. On creating this release, two workflows will start. One releases to pypi, and the other releases to conda.
- Validate these workflows pass, and the package is properly released on both platforms.

CHAPTER

ELEVEN

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.

11.1 [3.0.0] - 2021-02-23

11.1.1 Fixed

- Handled multi-word tokens better in Tree creation by simply ignoring them since they do not interact with dependency relations.
- Several linting and style issues on updating tools.
- Conllable metaclass is now set properly in Python 3 method rather than old Python 2 method.
- Several typos in the docstrings via new codespell linting process.
- Thorough dependency cleanup and updating.

11.1.2 Changed

- A Sentence with no root will throw a ValueError on Tree creation rather than returning an empty tree.
- Continued changes and improvements with DevOps moving from TravisCI to GitHub actions, having better structured tests, typo checking as part of linting, etc.
- Dependencies are now separated into build specific and other dev dependencies

11.1.3 Added

- Type annotations the public and internal API
- Added UD 2.7 to the list of versions validated against.
- Revamped version mechanism so that this is now present within the actual module under `pyconll.__version__` as a string

11.1.4 Removed

- Python 3.4 and 3.5 support was removed as part of supporting type annotations and staying up to date with the currently supported versions.
- Loading or iterating from network url was removed. It introduced a dependency that was better removed, and did not seem to be used often in the wild. It also encourages inefficient design and can be easily replicated for those who need it.

11.2 [2.3.3] - 2020-10-25

11.2.1 Fixed

- Github action workflows were using old version of python that was no longer supported.

11.3 [2.3.2] - 2020-10-25

11.3.1 Fixed

- General quality improvements including documentation improvements, docstring improvements, better testing strategies, etc.
- Clarified supported UD versions in README

11.4 [2.3.1] - 2020-10-06

11.4.1 Fixed

- PyPi workflow on release had improper repository url

11.5 [2.3] - 2020-10-06

11.5.1 Fixed

- Bug in outputting enhanced dependencies when index had a range or was for an empty node
- Typo in variable reference in documentation generation code
- Corrected docstring for `set_meta` for the Sentence API

11.5.2 Added

- `remove_meta` was added to the Sentence API thanks to alexeykosh

11.5.3 Changed

- Miscellaneous testing improvements and investments, Makefile improvements, release script improvements, and community improvements

11.6 [2.2.1] - 2019-11-17

11.6.1 Fixed

- Branding information typo within setup.py
- Spurious command in Makefile recipe

11.6.2 Added

- Added `python_requires` clause to setup.py to prevent installation on unsupported platforms
- Include information in README about `setuptools` version needed to properly package within `python_requires` information
- Conda packaging support along with information in README about new installation method

11.6.3 Changed

- `pyconll` version is now housed in .version file so that this version only needs to be changed in one place before release.

11.7 [2.2.0] - 2019-10-01

11.7.1 Changed

- Use slots on Token and Sentence class for more efficient memory usage with large amounts of objects
- Remove source fields on Token and Sentence. These were not an explicit part of the public API so this is not considered a breaking change.

11.8 [2.1.1] - 2019-09-04

11.8.1 Fixed

- Solved `math.inf` issue with python 3.4 where it does not exist

11.9 [2.1.0] - 2019-08-30

11.9.1 Fixed

- The example `reannotate\_ngrams.py` was out of sync with the function return type

11.9.2 Added

- `find_nonprojective_deps`` was added to look for non-projective dependencies within a sentence

11.10 [2.0.0] - 2019-05-09

11.10.1 Fixed

- `find_ngrams` in the `util` module did not properly match case insensitivity.
- `conllable` is now properly included in wildcard imports from `pyconll`.
- Issue when loading a CoNLL file over a network if the file contained UTF-8 characters. requests default assumes ASCII enconding on HTTP responses.
- The Token columns deps and feats were not properly sorted by attribute (either numeric index or case invariant lexicographic sort) on serialization

11.10.2 Changed

- Clearer and more consise documentation
- `find_ngrams` now returns the matched tokens as the last element of the yielded tuple.

11.10.3 Removed

- Document and paragraph ids on Sentences
- Line numbers on Tokens and Sentences
- Equality comparison on Tokens and Sentences. These types are mutable and implementing equality (with no hash overriding) causes issues for API clients.
- `SentenceTree` module. This functionaliy was moved to the Sentence class method `to_tree`.

11.10.4 Added

- `to_tree` method on `Sentence` that returns the Tree representing the Sentence dependency structure

11.10.5 Security

- Updates to `requirements.txt` to patch Jinja2 and requests

11.11 [1.1.4] - 2019-04-15

11.11.1 Fixed

- Parsing of underscore's for the form and lemma field, would automatically default to None, rather than the intended behavior.

11.12 [1.1.3] - 2019-01-03

11.12.1 Fixed

- When used on Windows, the default encoding of Windows-1252 was used when loading CoNLL-U files, however, CoNLL-U is UTF-8. This is now fixed.

11.13 [1.1.2] - 2018-12-28

11.13.1 Added

- *Getting Started* page on the documentation to make easier for newcomers

11.13.2 Fixed

- Versioning on docs page which had not been properly updated
- Some documentation errors
- `requests` version used in `requirements.txt` was insecure and updated to newer version

11.14 [1.1.1] - 2018-12-10

11.14.1 Fixed

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

11.15 [1.1.0] - 2018-11-11

11.15.1 Added

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

11.15.2 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

11.16 [1.0.1] - 2018-09-14

11.16.1 Changed

- Removed test packages from final shipped package.

11.17 [1.0] - 2018-09-13

11.17.1 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.

11.17.2 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.

11.18 [0.3.1] - 2018-08-08

11.18.1 Fixed

- Fixed issue with submodules not being packaged in build

11.19 [0.3] - 2018-07-28

11.19.1 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.

11.19.2 Fixed

- Some small bug fixes with parsing the token dicts.

11.20 [0.2.3] - 2018-07-23

11.20.1 Fixed

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

11.20.2 Added

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

11.21 [0.2.2] - 2018-07-18

11.21.1 Fixed

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

11.22 [0.2.1] - 2018-07-18

11.22.1 Fixed

- Installation issues when using `pip`

11.23 [0.2] - 2018-07-16

11.23.1 Added

- More documentation
- Util package for convenient and common logic

11.24 [0.1.1] - 2018-07-15

11.24.1 Added

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

11.25 [0.1] - 2018-07-04

11.25.1 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.

PYTHON MODULE INDEX

p

INDEX

Symbols

C

D

F

I

L

M

P

R

S

T

W