colonel Documentation

Release 1.2.0

The NLP Odyssey Authors

May 06, 2018

Contents

Colonel is a *Python 3* library for handling *CoNLL* data formats.

CHAPTER 1

colonel package

1.1 Subpackages

1.1.1 colonel.conllu package

Submodules

colonel.conllu.lexer module

Module providing the *ConlluLexerBuilder* class and related exception classes.

class colonel.conllu.lexer.**ConlluLexerBuilder**
Bases: object

Class containing *PLY Lex* rules for processing the *CoNLL-U* format and for creating new related *PLY* Lexer instances.

Usually you can simply invoke the class method *build()* which returns a *PLY* Lexer; such lexer instance is ready to process your input, making use of the rules provided by the *ConlluLexerBuilder* class itself.

classmethod build()
Returns a *PLY* Lexer instance for *CoNLL-U* processing.

The returned lexer makes use of the rules defined by *ConlluLexerBuilder*.

Return type Lexer

static find_column(*token*)
Given a LexToken, it returns the related column number.

Return type int

states = (('v0', 'exclusive'), ('v1', 'exclusive'), ('v2', 'exclusive'), ('v3', 'exclu

static t_ANY_error(*token*)

Return type None

static t_COMMENT(*token*)
[#][^n]*

Return type LexToken

static t_DECIMAL_ID(*token*)
([1-9][0-9]+|[0-9]).[1-9][0-9]*

Return type LexToken

t_INITIAL_v9_NEWLINE(*token*)
n

Return type LexToken

static t_INTEGER_ID(*token*)
[1-9][0-9]*

Return type LexToken

static t_RANGE_ID(*token*)
[1-9][0-9]*-[1-9][0-9]*

Return type LexToken

static t_c1_FORM(*token*)
[^nt]+

Return type LexToken

static t_c2_LEMMA(*token*)
[^nt]+

Return type LexToken

static t_c3_UPOS(*token*)

Return type LexToken

static t_c4_XPOS(*token*)
[^nt]+

Return type LexToken

static t_c5_FEATS(*token*)

Return type LexToken

static t_c6_HEAD(*token*)
([1-9][0-9]+|[0-9])|_

Return type LexToken

static t_c7_DEPREL(*token*)
[^nt]+

Return type LexToken

static t_c8_DEPS(*token*)

Return type LexToken

static t_c9_MISC(*token*)
[^nt]+

Return type LexToken

t_v0_v1_v2_v3_v4_v5_v6_v7_v8_TAB(*token*)
t

Return type LexToken

tokens = ('NEWLINE', 'TAB', 'COMMENT', 'INTEGER_ID', 'RANGE_ID', 'DECIMAL_ID', 'FORM',

exception colonel.conllu.lexer.**IllegalCharacterError**(*token*)
Bases: *colonel.conllu.lexer.LexerError*

Exception raised by *ConlluLexerBuilder* when a lexer error caused by invalid input is encountered.

An exception instance must be initialized with the LexToken which the lexer was not able to process, so that *line_number* and *column_number* can be extracted; a short error message is also generated by the constructor.

column_number = None
Column position, associated with *line_number*, containing the illegal character, or the start of an illegal sequence.

line_number = None
Line number containing the illegal character, or the start of an illegal sequence.

exception colonel.conllu.lexer.**LexerError**
Bases: Exception

Generic error class for *ConlluLexerBuilder*.

colonel.conllu.parser module

Module providing the *ConlluParserBuilder* class and related exception classes.

class colonel.conllu.parser.**ConlluParserBuilder**
Bases: object

Class containing *PLY Yacc* rules for processing the *CoNLL-U* format and for creating new related *PLY* LRParser instances.

Usually you can simply invoke the class method *build()* which returns a *PLY* LRParser; such parser instance is ready to process your input, making use of the rules provided by the *ConlluParserBuilder* class itself.

As usual, this class is paired with an associated lexer, which in in this case is served by *ConlluLexerBuilder*.

classmethod build()
Returns a *PLY* LRParser instance for *CoNLL-U* processing.

The returned parser makes use of the rules defined by *ConlluParserBuilder*.

Return type LRParser

static p_comment(*prod*)
comment : COMMENT NEWLINE

Return type None

static p_comments_many(*prod*)
comments : comments comment

Return type None

static p_comments_one(*prod*)
comments : comment

Return type None

static p_error(*token*)

Return type None

static p_sentence_with_comments(*prod*)
sentence : comments wordlines NEWLINE

Return type None

static p_sentence_without_comments(*prod*)
sentence : wordlines NEWLINE

Return type None

static p_sentences_many(*prod*)
sentences : sentences sentence

Return type None

static p_sentences_one(*prod*)
sentences : sentence

Return type None

static p_wordline_emptynode(*prod*)
wordline : DECIMAL_ID TAB FORM TAB LEMMA TAB UPOS TAB XPOS TAB FEATS TAB HEAD TAB DEPREL TAB DEPS TAB MISC NEWLINE

Return type None

static p_wordline_multiword(*prod*)
wordline : RANGE_ID TAB FORM TAB LEMMA TAB UPOS TAB XPOS TAB FEATS TAB HEAD TAB DEPREL TAB DEPS TAB MISC NEWLINE

Return type None

static p_wordline_word(*prod*)
wordline : INTEGER_ID TAB FORM TAB LEMMA TAB UPOS TAB XPOS TAB FEATS TAB HEAD TAB DEPREL TAB DEPS TAB MISC NEWLINE

Return type None

static p_wordlines_many(*prod*)
wordlines : wordlines wordline

Return type None

static p_wordlines_one(*prod*)
wordlines : wordline

Return type None

exception colonel.conllu.parser.**IllegalEmptyNodeError**(*prod*)
Bases: *colonel.conllu.parser.ParserError*

Exception raised by *ConlluParserBuilder* when a word line was parsed correctly and has been recognised as an *empty node* line, however the data is not valid for this kind of element.

An exception instance must be initialized with the YaccProduction related to the word line containing illegal data, so that the line_number can be extracted; a short error message is also generated by the constructor.

exception colonel.conllu.parser.**IllegalEofError**
Bases: *colonel.conllu.parser.ParserError*

Exception raised by *ConlluParserBuilder* when a parser error caused by invalid *end-of-file* is encountered.

When this exception is raised, it means that the end of the input data has been reached, but some additional tokens were expected in order to be valid *CoNLL-U*.

exception colonel.conllu.parser.**IllegalMultiwordError**(*prod*)
Bases: *colonel.conllu.parser.ParserError*

Exception raised by *ConlluParserBuilder* when a word line was parsed correctly and has been recognised as a *multiword token* line, however the data is not valid for this kind of element.

An exception instance must be initialized with the `YaccProduction` related to the word line containing illegal data, so that the `line_number` can be extracted; a short error message is also generated by the constructor.

exception colonel.conllu.parser.**IllegalTokenError**(*t*)
Bases: *colonel.conllu.parser.ParserError*

Exception raised by *ConlluParserBuilder* when a parser error caused by invalid token is encountered.

An exception instance must be initialized with the `LexToken` which the parser was not able to process, so that all the exception attributes can be extracted; a short error message is also generated by the constructor.

column_number = None
Column position, associated with *line_number*, related to the illegal token encountered, or to the first token of an illegal tokens sequence.

line_number = None
Line number related to the illegal token encountered, or to the first token of an illegal tokens sequence.

type = None
The type of the illegal token encountered, or of the first token of an illegal tokens sequence.

value = None
The value of the illegal token encountered, or of the first token of an illegal tokens sequence.

exception colonel.conllu.parser.**ParserError**
Bases: `Exception`

Generic error class for *ConlluParserBuilder*.

Module contents

This package provides methods and modules to process the *CoNLL-U* format.

In most situations it's sufficient to make use of *parse()* and *to_conllu()* functions, without caring too much about the implementation under the hood.

In more detail, this package provides a lexical analyzer (see *lexer*) and a parser (see *parser*) to transform the raw string input into related *Sentence* objects.

Lexer and parser classes are implemented taking advantage of the *PLY (Python Lex-Yacc)* library; you can learn more from the PLY documentation and from the Lex & Yacc Page.

colonel.conllu.**parse**(*content*)
Parses a *CoNLL-U* string content, returning a list of sentences.

Raises

- ***lexer.LexerError*** – (any specific subclass) in case of invalid input breaking the rules of the *CoNLL-U* lexer
- ***parser.ParserError*** – (any specific subclass) in case of invalid input breaking the rules of the *CoNLL-U* parser

Parameters **content** (str) – *CoNLL-U* formatted string to be parsed

Return type List[*Sentence*]

Returns list of parsed *Sentence* items

colonel.conllu.**to_conllu**(*sentences*)

Serializes a list of sentences to a formatted *CoNLL-U* string.

This method simply concatenates the output of *Sentence.to_conllu()* for each given sentence and do not perform any validity check; sentences and elements not compatible with *CoNLL-U* format could lead to an incorrect output value or raising of exceptions.

Parameters **sentences** (List[*Sentence*]) – list of *Sentence* items

Return type str

Returns a *CoNLL-U* formatted representation of the sentences

1.2 Submodules

1.2.1 colonel.base_rich_sentence_element module

Module providing the *BaseRichSentenceElement* class.

class colonel.base_rich_sentence_element.**BaseRichSentenceElement**(*lemma=None*, *upos=None*, *xpos=None*, *feats=None*, *deps=None*, ***kwargs*)

Bases: *colonel.base_sentence_element.BaseSentenceElement*

Abstract class containing basic information in common with some specific elements being part of a sentence.

It is compliant with the *CoNLL-U* format, in the sense that it provides a common foundation for elements of type *word* and *empty nodes*, which can be made up of a richer set of fields in comparison to other elements, such as the *(multiword) tokens*.

deps

Enhanced dependency graph, usually in the form of a list of head-deprel pairs.

It is compatible with *CoNLL-U* DEPS field.

You are free to assign to it any kind of value suitable for your project.

feats

List of morphological features from the universal feature inventory or from a defined language-specific extension.

It is compatible with *CoNLL-U* FEATS field.

You are free to assign to it any kind of value suitable for your project.

is_valid()
Returns whether or not the object can be considered valid, however ignoring the context of the sentence in which the word itself is possibly inserted.

An instance of type *BaseRichSentenceElement* is *always* considered valid, independently from any value of its attributes (it doesn't provide any additional check to the overridden superclass method).

lemma
Lemma of the element.

It is compatible with *CoNLL-U* LEMMA field.

to_conllu()
Returns a *CoNLL-U* formatted representation of the element.

This method is expected to be overridden by each specific element.

upos
Universal part-of-speech tag.

It is compatible with *CoNLL-U* UPOS field.

xpos
Language-specific part-of-speech tag.

It is compatible with *CoNLL-U* XPOS field.

1.2.2 colonel.base_sentence_element module

Module providing the *BaseSentenceElement* class.

class colonel.base_sentence_element.**BaseSentenceElement**(*form=None*, *misc=None*)
Bases: object

Abstract class containing the minimum information in common with all specific elements being part of a sentence.

In the context of this library, it is expected that each item of a sentence is an instance of a *BaseSentenceElement* subclass.

The generic term *element* is used in order to prevent confusion, while each specialized element (i.e. a subclass of *BaseSentenceElement*) will adopt a more appropriate naming convention, so that, for example, a sentence will be usually formed by *words*, *tokens* or *nodes*.

form
Word form or punctuation symbol.

It is compatible with *CoNLL-U* FORM field.

is_valid()
Returns whether or not the object can be considered valid, however ignoring the context of the sentence in which the word itself is possibly inserted.

An instance of type BaseWord is *always* considered valid, independently from any value of its attributes.

Return type bool

misc
Any other annotation.

It is compatible with *CoNLL-U* MISC field.

to_conllu()
Returns a *CoNLL-U* formatted representation of the element.

This method is expected to be overridden by each specific element.

1.2.3 colonel.emptynode module

Module providing the *EmptyNode* class.

class colonel.emptynode.**EmptyNode**(*main_index=None*, *sub_index=None*, ***kwargs*)
Bases: *colonel.base_rich_sentence_element.BaseRichSentenceElement*

Representation of an *Empty Node* sentence element

is_valid()
Returns whether or not the object can be considered valid, however ignoring the context of the sentence in which the word itself is possibly inserted.

In compliance with the *CoNLL-U* format, an instance of type *EmptyNode* is considered valid only when *main_index* is set to a value equal to or greater than zero (0) **and** *sub_index* is set to a value greater than zero (0).

Return type bool

main_index
The primary index of the empty node.

This usually corresponds to the value of the *Word.index* after which the empty node is inserted, or to zero (0) if the empty node is inserted before the first word of the sentence (the one with *index* equal to 1).

It is compatible with *CoNLL-U* ID field, which in case of an empty node is a decimal number: the *main index* here corresponds to the integer part of such value.

sub_index
The secondary index of the empty node.

It is compatible with *CoNLL-U* ID field, which in case of an empty node is a decimal number: the *sub index* here corresponds to the decimal part of such value.

to_conllu()
Returns a *CoNLL-U* formatted representation of the element.

No validity check is performed on the attributes; values not compatible with *CoNLL-U* format could lead to an incorrect output value or raising of exceptions.

Return type str

1.2.4 colonel.multiword module

Module providing the *Multiword* class.

class colonel.multiword.**Multiword**(*first_index=None*, *last_index=None*, ***kwargs*)
Bases: *colonel.base_sentence_element.BaseSentenceElement*

Representation of a *Multiword Token* sentence element

first_index
The first word index (inclusive) covered by the multiword token.

This usually corresponds to the value of the *Word.index* of the first *Word* which is part of this multiword token.

It is compatible with *CoNLL-U* `ID` field, which in case of a multiword token is a range of integer numbers, where first and last bound indexes are separated by a dash (`-`): the first index here corresponds to the value at left.

is_valid()
Returns whether or not the object can be considered valid, however ignoring the context of the sentence in which the word itself is possibly inserted.

In compliance with the *CoNLL-U* format, an instance of type *Multiword* is considered valid only when *first_index* is set to a value greater than zero (`0`) **and** *last_index* is set to a value greater than *first_index*.

Return type bool

last_index
The last word index (inclusive) covered by the multiword token.

This usually corresponds to the value of the *Word.index* of the last *Word* which is part of this multiword token.

It is compatible with *CoNLL-U* `ID` field, which in case of a multiword token is a range of integer numbers, where first and last bound indexes are separated by a dash (`-`): the first index here corresponds to the value at right.

to_conllu()
Returns a *CoNLL-U* formatted representation of the element.

No validity check is performed on the attributes; values not compatible with *CoNLL-U* format could lead to an incorrect output value or raising of exceptions.

Return type str

1.2.5 colonel.sentence module

Module providing the *Sentence* class.

class colonel.sentence.**Sentence**(*elements=None*, *comments=None*)
Bases: object

Representation of a *sentence*.

This class is modeled starting from the *CoNLL-U Format* specification, which states that *sentences consist of one or more word lines*. Each *word line* contains a series of fields, first of all an *ID*, the value of which determines the *kind* of the whole line: a *single word*, a *(multiword) token* or an *empty node*.

Analogously, here a *Sentence* mostly consists of an ordered list of *elements*, which can be object of any *BaseSentenceElement*'s subclass, commonly a *Word*, a *Multiword* or an *EmptyNode*.

Since the *CoNLL-U* format allows the presence of comment lines before a sentence, the *comments* attribute is made available here as a simple list of strings.

comments
Miscellaneous comments related to the sentence.

For the time being, in the context of this project no particular meaning is given to the values of this attribute, however the following guidelines *should* be followed in order to facilitate possible future usages and processing:

- the presence of the leading # character (which denotes the start of a comment line in *CoNLL-U* format) is discouraged, in order to keep comments more format-independent;
- each comment should be always stripped from leading/trailing spaces or newline characters.

elements
Ordered list of words, tokens and nodes which form the sentence.

Usually this list can be freely and directly manipulated, since the methods of the class always recompute their returned value accordingly; just pay particular attention performing changes while in the context of iterations (see for example *words()* and *raw_tokens()* methods).

is_valid()
Returns whether or not the sentence is valid.

The checks implemented here are mostly based on the *CoNLL-U* format and on the most widely adopted common practices among NLP and dependency parsing contexts, yet including a minimum set of essential validation, so that you are free to use this as a foundation for other custom rules in your application.

A sentence is considered *valid* only if **all** of the following conditions apply:

- there is at least one element of type `Word`;
- every single element is valid as well - see *BaseSentenceElement.is_valid()* and the overriding of its subclasses;
- the ordered sequence of the elements and their *ID* is valid, that is:
  - the sequence of *Word.index* starts from `1` and progressively increases by 1 step;
  - there are no *index* duplicates or range overlapping;
  - the *EmptyNode* elements (if any) are correctly placed after the *Word* element related to their *EmptyNode.main_index* (or before the first word of the sentence, when the *main index* is zero), and for each sequence of *empty nodes* their *EmptyNode.sub_index* starts from `1` and progressively increases by 1 step;
  - the *Multiword* elements (if any) are correctly placed before the first *Word* included in their *index* range, and each range always cover existing *Word* elements in the sentence;
- if one or more *Word.head* values are set (not `None`), each head must refer to the *index* of a *Word* existing within the sentence, or at least be equal to zero (`0`, for `root` grammatical relations).

Return type `bool`

raw_tokens()
Extracts the raw token sequence.

Iterates through *elements* and yields the only elements which represent the raw sequence of tokens in the sentence. The result includes *Word* and *Multiword* elements, skipping all *Word* items which indexes are included in the range of a preceding `MultiWord`.

Empty nodes are ignored.

This method do not perform any validity check among the elements, so if you want to ensure valid and meaningful results, please refer to *is_valid()*; unless you really know what you are doing, iterating an invalid sentence could lead to wrong or incoherent results or unexpected behaviours.

Return type `Iterator`[`Union`[*Word*, *Multiword*]]

to_conllu()
Returns a *CoNLL-U* formatted representation of the sentence.

No validity check is performed on the sentence and its element; elements and values not compatible with *CoNLL-U* format could lead to an incorrect output value or raising of exceptions.

Return type `str`

words()
Extracts the sequence of words.

Iterates through *elements* and yields *Word* elements only. This can be especially handy in many dependency parsing contexts, where the focus mostly resides among simple words and their relations, ignoring the additional information carried by *empty nodes* and *(multiword) tokens*.

This method do not perform any validity check among the elements, so if you want to ensure valid and meaningful results, please refer to *is_valid()*; unless you really know what you are doing, iterating an invalid sentence could lead to wrong or incoherent results or unexpected behaviours.

Return type Iterator[*Word*]

1.2.6 colonel.upostag module

Module providing the *UposTag* enumeration.

class colonel.upostag.**UposTag**
Bases: enum.Enum

Enumeration of *Universal POS tags*.

These tags mark the core part-of-speech categories according to the *Universal Dependencies* framework.

See also the UPOS field in the *CoNLL-U* format.

Note: always refer to the name of each member; values are automatically generated and thus **MUST** be considered opaque.

ADJ = 1
adjective

ADP = 2
adposition

ADV = 3
adverb

AUX = 4
auxiliary

CCONJ = 5
coordinating conjunction

DET = 6
determiner

INTJ = 7
interjection

NOUN = 8
noun

NUM = 9
numeral

PART = 10
particle

PRON = 11
pronoun

PROPN = 12
proper noun

PUNCT = 13
punctuation

SCONJ = 14
subordinating conjunction

SYM = 15
symbol

VERB = 16
verb

X = 17
other

1.2.7 colonel.word module

Module providing the *Word* class.

class colonel.word.**Word**(*index=None*, *head=None*, *deprel=None*, ***kwargs*)
Bases: *colonel.base_rich_sentence_element.BaseRichSentenceElement*

Representation of a *Word* sentence element

deprel
Universal dependency relation to the *head* or a defined language-specific subtype of one.

It is compatible with *CoNLL-U* DEPREL field.

head
Head of the current word, which is usually a value of another Word's *index* or zero (0, for root grammatical relations).

It is compatible with *CoNLL-U* HEAD field.

index
Word index.

It is compatible with *CoNLL-U* ID field.

The term *index* has been preferred over the more conventional *ID*, mostly for the purpose of preventing confusion, especially with Python's id() built-in function (which returns the *"identity"* of an object).

is_valid()
Returns whether or not the object can be considered valid, however ignoring the context of the sentence in which the word itself is possibly inserted.

In compliance with the *CoNLL-U* format, an instance of type *Word* is considered valid only when *index* is set to a value greater than zero (0).

Return type bool

to_conllu()
Returns a *CoNLL-U* formatted representation of the element.

No validity check is performed on the attributes; values not compatible with *CoNLL-U* format could lead to an incorrect output value or raising of exceptions.

Return type str

1.3 Module contents

Colonel - a Python 3 library for handling CoNLL data formats

class colonel.**Sentence**(*elements=None*, *comments=None*)
Bases: object

Representation of a *sentence*.

This class is modeled starting from the *CoNLL-U Format* specification, which states that *sentences consist of one or more word lines*. Each *word line* contains a series of fields, first of all an *ID*, the value of which determines the *kind* of the whole line: a *single word*, a *(multiword) token* or an *empty node*.

Analogously, here a *Sentence* mostly consists of an ordered list of *elements*, which can be object of any *BaseSentenceElement*'s subclass, commonly a *Word*, a *Multiword* or an *EmptyNode*.

Since the *CoNLL-U* format allows the presence of comment lines before a sentence, the *comments* attribute is made available here as a simple list of strings.

comments
Miscellaneous comments related to the sentence.

For the time being, in the context of this project no particular meaning is given to the values of this attribute, however the following guidelines *should* be followed in order to facilitate possible future usages and processing:

- the presence of the leading # character (which denotes the start of a comment line in *CoNLL-U* format) is discouraged, in order to keep comments more format-independent;
- each comment should be always stripped from leading/trailing spaces or newline characters.

elements
Ordered list of words, tokens and nodes which form the sentence.

Usually this list can be freely and directly manipulated, since the methods of the class always recompute their returned value accordingly; just pay particular attention performing changes while in the context of iterations (see for example *words()* and *raw_tokens()* methods).

is_valid()
Returns whether or not the sentence is valid.

The checks implemented here are mostly based on the *CoNLL-U* format and on the most widely adopted common practices among NLP and dependency parsing contexts, yet including a minimum set of essential validation, so that you are free to use this as a foundation for other custom rules in your application.

A sentence is considered *valid* only if **all** of the following conditions apply:

- there is at least one element of type *Word*;
- every single element is valid as well - see *BaseSentenceElement.is_valid()* and the overriding of its subclasses;
- the ordered sequence of the elements and their *ID* is valid, that is:
  - the sequence of *Word.index* starts from 1 and progressively increases by 1 step;
  - there are no *index* duplicates or range overlapping;
  - the *EmptyNode* elements (if any) are correctly placed after the *Word* element related to their *EmptyNode.main_index* (or before the first word of the sentence, when the *main index* is zero), and for each sequence of *empty nodes* their *EmptyNode.sub_index* starts from 1 and progressively increases by 1 step;

  - the *Multiword* elements (if any) are correctly placed before the first *Word* included in their *index* range, and each range always cover existing *Word* elements in the sentence;

- if one or more *Word.head* values are set (not `None`), each head must refer to the *index* of a *Word* existing within the sentence, or at least be equal to zero (`0`, for `root` grammatical relations).

Return type `bool`

raw_tokens()
Extracts the raw token sequence.

Iterates through *elements* and yields the only elements which represent the raw sequence of tokens in the sentence. The result includes *Word* and *Multiword* elements, skipping all *Word* items which indexes are included in the range of a preceding `MultiWord`.

Empty nodes are ignored.

This method do not perform any validity check among the elements, so if you want to ensure valid and meaningful results, please refer to *is_valid()*; unless you really know what you are doing, iterating an invalid sentence could lead to wrong or incoherent results or unexpected behaviours.

Return type `Iterator`[`Union`[*Word*, *Multiword*]]

to_conllu()
Returns a *CoNLL-U* formatted representation of the sentence.

No validity check is performed on the sentence and its element; elements and values not compatible with *CoNLL-U* format could lead to an incorrect output value or raising of exceptions.

Return type `str`

words()
Extracts the sequence of words.

Iterates through *elements* and yields *Word* elements only. This can be especially handy in many dependency parsing contexts, where the focus mostly resides among simple words and their relations, ignoring the additional information carried by *empty nodes* and *(multiword) tokens*.

This method do not perform any validity check among the elements, so if you want to ensure valid and meaningful results, please refer to *is_valid()*; unless you really know what you are doing, iterating an invalid sentence could lead to wrong or incoherent results or unexpected behaviours.

Return type `Iterator`[*Word*]

class colonel.**Word**(*index=None*, *head=None*, *deprel=None*, ***kwargs*)
Bases: *colonel.base_rich_sentence_element.BaseRichSentenceElement*

Representation of a *Word* sentence element

deprel
Universal dependency relation to the *head* or a defined language-specific subtype of one.

It is compatible with *CoNLL-U* `DEPREL` field.

head
Head of the current word, which is usually a value of another Word's *index* or zero (`0`, for `root` grammatical relations).

It is compatible with *CoNLL-U* `HEAD` field.

index
Word index.

It is compatible with *CoNLL-U* `ID` field.

The term *index* has been preferred over the more conventional *ID*, mostly for the purpose of preventing confusion, especially with Python's `id()` built-in function (which returns the *"identity"* of an object).

is_valid()
Returns whether or not the object can be considered valid, however ignoring the context of the sentence in which the word itself is possibly inserted.

In compliance with the *CoNLL-U* format, an instance of type *Word* is considered valid only when *index* is set to a value greater than zero (`0`).

Return type `bool`

to_conllu()
Returns a *CoNLL-U* formatted representation of the element.

No validity check is performed on the attributes; values not compatible with *CoNLL-U* format could lead to an incorrect output value or raising of exceptions.

Return type `str`

class colonel.**EmptyNode**(*main_index=None*, *sub_index=None*, ***kwargs*)
Bases: *colonel.base_rich_sentence_element.BaseRichSentenceElement*

Representation of an *Empty Node* sentence element

is_valid()
Returns whether or not the object can be considered valid, however ignoring the context of the sentence in which the word itself is possibly inserted.

In compliance with the *CoNLL-U* format, an instance of type *EmptyNode* is considered valid only when *main_index* is set to a value equal to or greater than zero (`0`) **and** *sub_index* is set to a value greater than zero (`0`).

Return type `bool`

main_index
The primary index of the empty node.

This usually corresponds to the value of the *Word.index* after which the empty node is inserted, or to zero (`0`) if the empty node is inserted before the first word of the sentence (the one with *index* equal to `1`).

It is compatible with *CoNLL-U* `ID` field, which in case of an empty node is a decimal number: the *main index* here corresponds to the integer part of such value.

sub_index
The secondary index of the empty node.

It is compatible with *CoNLL-U* `ID` field, which in case of an empty node is a decimal number: the *sub index* here corresponds to the decimal part of such value.

to_conllu()
Returns a *CoNLL-U* formatted representation of the element.

No validity check is performed on the attributes; values not compatible with *CoNLL-U* format could lead to an incorrect output value or raising of exceptions.

Return type `str`

class colonel.**Multiword**(*first_index=None*, *last_index=None*, ***kwargs*)
Bases: *colonel.base_sentence_element.BaseSentenceElement*

Representation of a *Multiword Token* sentence element

first_index
The first word index (inclusive) covered by the multiword token.

This usually corresponds to the value of the *Word.index* of the first *Word* which is part of this multiword token.

It is compatible with *CoNLL-U* ID field, which in case of a multiword token is a range of integer numbers, where first and last bound indexes are separated by a dash (−): the first index here corresponds to the value at left.

is_valid()
Returns whether or not the object can be considered valid, however ignoring the context of the sentence in which the word itself is possibly inserted.

In compliance with the *CoNLL-U* format, an instance of type *Multiword* is considered valid only when *first_index* is set to a value greater than zero (0) **and** *last_index* is set to a value greater than *first_index*.

Return type bool

last_index
The last word index (inclusive) covered by the multiword token.

This usually corresponds to the value of the *Word.index* of the last *Word* which is part of this multiword token.

It is compatible with *CoNLL-U* ID field, which in case of a multiword token is a range of integer numbers, where first and last bound indexes are separated by a dash (−): the first index here corresponds to the value at right.

to_conllu()
Returns a *CoNLL-U* formatted representation of the element.

No validity check is performed on the attributes; values not compatible with *CoNLL-U* format could lead to an incorrect output value or raising of exceptions.

Return type str

class colonel.**UposTag**
Bases: enum.Enum

Enumeration of *Universal POS tags*.

These tags mark the core part-of-speech categories according to the *Universal Dependencies* framework.

See also the UPOS field in the *CoNLL-U* format.

Note: always refer to the name of each member; values are automatically generated and thus **MUST** be considered opaque.

ADJ = 1
adjective

ADP = 2
adposition

ADV = 3
adverb

AUX = 4
auxiliary

CCONJ = 5
coordinating conjunction

DET = 6
determiner

INTJ = 7
interjection

NOUN = 8
noun

NUM = 9
numeral

PART = 10
particle

PRON = 11
pronoun

PROPN = 12
proper noun

PUNCT = 13
punctuation

SCONJ = 14
subordinating conjunction

SYM = 15
symbol

VERB = 16
verb

X = 17
other

CHAPTER 2

Python Module Index

CHAPTER 3

Alphabetical Index

Python Module Index

C

Index

L

M

N

P

R

S

T

U

V

W

X