corpy

Release 0.2.3

David Lukeš

Aug 20, 2019

USER GUIDES

CHAPTER

ONE

INSTALLATION

```
$ pip3 install corpy
```

Only recent versions of Python 3 (3.6+) are supported by design.

CHAPTER

TWO

WHAT IS CORPY?

A fancy plural for *corpus* ;) Also, a collection of handy but not especially mutually integrated tools for dealing with linguistic data. It abstracts away functionality which is often needed in practice for teaching and/or day to day work at the Czech National Corpus, without aspiring to be a fully featured or consistent NLP framework.

The short URL to the docs is: https://corpy.rtfd.io/

Here's an idea of what you can do with CorPy:

- add linguistic annotation to raw textual data using either UDPipe or MorphoDiTa

---

Note: Should I pick UDPipe or MorphoDiTa?

UDPipe is the successor to MorphoDiTa, extending and improving upon the original codebase. It has more features at the cost of being somewhat more complex: it does both morphological tagging (including lemmatization) and syntactic parsing, and it handles a number of different input and output formats. You can also download pre-trained models for many different languages.

By contrast, MorphoDiTa only has pre-trained models for Czech and English, and only performs morphological tagging (including lemmatization). However, its output is more straightforward – it just splits your text into tokens and annotates them, whereas UDPipe can (depending on the model) introduce additional tokens necessary for a more explicit analysis, add multi-word tokens etc. This is because UDPipe is tailored to the type of linguistic analysis conducted within the UniversalDependencies project, using the CoNLL-U data format.

MorphoDiTa can also help you if you just want to tokenize text and don't have a language model available.

- easily generate word clouds
- generate phonetic transcripts of Czech texts
- wrangle corpora in the vertical format devised originally for CWB, used also by (No)SketchEngine
- plus some command line utilities

2.1 Tag and parse text with UDPipe

NOTE: When playing around with UDPipe interactively, it's highly recommended to use IPython or a Jupyter notebook. You'll automatically get nice pretty-printing.

2.1.1 Overview

UDPipe is a fast and convenient library for stochastic morphological tagging (including lemmatization) and syntactic parsing of text. The `corpy.udpipe` module aims to give easy access to the most commonly used features of the

library; for more advanced use cases, including if you need speedups in performance critical code, you might need to use the more lower-level ufal.udpipe package, on top of which this module is built.

In order to use UDPipe, you need a pre-trained model for your language of interest. Models are available for many languages, for more information, refer to the UDPipe website. **When using the models, please make sure to respect their CC BY-NC-SA license!**

In order to better understand how UDPipe represents tagged and parsed text, it is useful to familiarize yourself with the CoNLL-U data format. UDPipe data structures (sentences, words, multi-word tokens, empty nodes, comments) map onto concepts defined in this format.

In addition to this guide, there is also an *API reference* for *corpy.udpipe*. For an overview of the API of underlying `ufal.udpipe` objects (listing available attributes and methods), see here.

2.1.2 Processing text

Tagging and parsing text using UDPipe is fairly simple. Just load a UDPipe *Model*:

```
>>> from corpy.udpipe import Model
>>> m = Model("./czech-pdt-ud-2.4-190531.udpipe")
```

And process some text using the *process()* method (the method creates a generator, so you'll need e.g. `list()` to tease all of the elements out of it):

```
>>> sents = list(m.process("Je zima. Bude sněžit."))
>>> sents
[<Swig Object of type 'sentence *' at 0x...>, <Swig Object of type 'sentence *' at 0x.
↪..>]
```

Ouch. This output is not really helpful. This is why it's recommended to use IPython or Jupyter, because at a regular Python REPL, the output of UDPipe is rendered as opaque Swig objects.

However, if the IPython package is at least installed, you can explicitly pretty-print the output using the *pprint()* function:

```
>>> from corpy.udpipe import pprint
>>> pprint(sents)
[Sentence(
   comments=['# newdoc', '# newpar', '# sent_id = 1', '# text = Je zima.'],
   words=[
     Word(id=0, <root>),
     Word(id=1,
          form='Je',
          lemma='být',
          xpostag='VB-S---3P-AA---',
          upostag='VERB',
          feats=
↪'Mood=Ind|Number=Sing|Person=3|Polarity=Pos|Tense=Pres|VerbForm=Fin|Voice=Act',
          head=0,
          deprel='root'),
     Word(id=2,
          form='zima',
          lemma='zima',
          xpostag='NNFS1-----A----',
          upostag='NOUN',
          feats='Case=Nom|Gender=Fem|Number=Sing|Polarity=Pos',
          head=1,
```

(continues on next page)

(continued from previous page)

```
          deprel='nsubj',
          misc='SpaceAfter=No'),
    Word(id=3,
          form='.',
          lemma='.',
          xpostag='Z:-------------',
          upostag='PUNCT',
          head=1,
          deprel='punct')]),
Sentence(
  comments=['# sent_id = 2', '# text = Bude sněžit.'],
  words=[
    Word(id=0, <root>),
    Word(id=1,
          form='Bude',
          lemma='být',
          xpostag='VB-S---3F-AA---',
          upostag='AUX',
          feats=
↪'Mood=Ind|Number=Sing|Person=3|Polarity=Pos|Tense=Fut|VerbForm=Fin|Voice=Act',
          head=2,
          deprel='aux'),
    Word(id=2,
          form='sněžit',
          lemma='sněžit',
          xpostag='Vf--------A----',
          upostag='VERB',
          feats='Aspect=Imp|Polarity=Pos|VerbForm=Inf',
          head=0,
          deprel='root',
          misc='SpaceAfter=No'),
    Word(id=3,
          form='.',
          lemma='.',
          xpostag='Z:-------------',
          upostag='PUNCT',
          head=2,
          deprel='punct',
          misc='SpaceAfter=No')])]
```

Much better! And again, calling `pprint(sents)` is not necessary when using IPython or Jupyter, you can just evaluate `sents` and it will be pretty-printed automatically.

2.1.3 Pretty-printing options

The output of UDPipe can be quite verbose – the individual objects have many fields. However, some values are not really that interesting (e.g. the empty string for string attributes, or `-1` for integer attributes). Therefore, they are hidden by the pretty-printer by default, so as to make the output more concise.

Sometimes though, you might want exhaustive pretty-printing, e.g. to learn about all of the possible attributes, even though your output doesn't happen to have any useful values in them. In order to do that, disable the `digest` option using the *`pprint_config()`* function:

```
>>> from corpy.udpipe import pprint_config
>>> pprint_config(digest=False)
```

(continues on next page)

(continued from previous page)

```
>>> pprint(sents)
[Sentence(
   comments=['# newdoc', '# newpar', '# sent_id = 1', '# text = Je zima.'],
   words=[
     Word(id=0,
          form='<root>',
          lemma='<root>',
          xpostag='<root>',
          upostag='<root>',
          feats='<root>',
          head=-1,
          deprel='',
          deps='',
          misc=''),
     Word(id=1,
          form='Je',
          lemma='být',
          xpostag='VB-S---3P-AA---',
          upostag='VERB',
          feats=
↪'Mood=Ind|Number=Sing|Person=3|Polarity=Pos|Tense=Pres|VerbForm=Fin|Voice=Act',
          head=0,
          deprel='root',
          deps='',
          misc=''),
     Word(id=2,
          form='zima',
          lemma='zima',
          xpostag='NNFS1-----A----',
          upostag='NOUN',
          feats='Case=Nom|Gender=Fem|Number=Sing|Polarity=Pos',
          head=1,
          deprel='nsubj',
          deps='',
          misc='SpaceAfter=No'),
     Word(id=3,
          form='.',
          lemma='.',
          xpostag='Z:--------------',
          upostag='PUNCT',
          feats='',
          head=1,
          deprel='punct',
          deps='',
          misc='')],
   multiwordTokens=[],
   emptyNodes=[]),
 Sentence(
   comments=['# sent_id = 2', '# text = Bude sněžit.'],
   words=[
     Word(id=0,
          form='<root>',
          lemma='<root>',
          xpostag='<root>',
          upostag='<root>',
          feats='<root>',
          head=-1,
```

(continues on next page)

(continued from previous page)

```
          deprel='',
          deps='',
          misc=''),
     Word(id=1,
          form='Bude',
          lemma='být',
          xpostag='VB-S---3F-AA---',
          upostag='AUX',
          feats=
↪'Mood=Ind|Number=Sing|Person=3|Polarity=Pos|Tense=Fut|VerbForm=Fin|Voice=Act',
          head=2,
          deprel='aux',
          deps='',
          misc=''),
     Word(id=2,
          form='sněžit',
          lemma='sněžit',
          xpostag='Vf--------A----',
          upostag='VERB',
          feats='Aspect=Imp|Polarity=Pos|VerbForm=Inf',
          head=0,
          deprel='root',
          deps='',
          misc='SpaceAfter=No'),
     Word(id=3,
          form='.',
          lemma='.',
          xpostag='Z:-------------',
          upostag='PUNCT',
          feats='',
          head=2,
          deprel='punct',
          deps='',
          misc='SpaceAfter=No')],
   multiwordTokens=[],
   emptyNodes=[])]
```

Let's turn digest back on to save space below.

```
>>> pprint_config(digest=True)
```

2.1.4 Input and output formats

UDPipe supports a variety of input and output formats. For convenience, they are listed in the documentation of the *corpy.udpipe.Model.process()* method, but the most up-to-date, reference list is always available in the UDPipe API docs.

One format which is particularly useful is the CoNLL-U format: it's the format of the UniversalDependencies project, and as such, it's intimately associated with UDPipe, which is also part of the project. Reading up on the CoNLL-U format can help you better understand how UDPipe represents tagged and parsed text, especially some of the less straightforward features (e.g. multi-word tokens and empty nodes).

Say you have a small two-sentence corpus in the "horizontal" format (one sentence per line, words separated by spaces), and you want to tag it, parse it, and output it in the CoNLL-U format. You can do it like so:

```
>>> horizontal = """Je zima .
... Bude sněžit ."""
>>> conllu_sents = list(m.process(horizontal, in_format="horizontal", out_format=
↪"conllu"))
>>> conllu_sents
['# newdoc\n# newpar\n# sent_id = 1\n1\tJe\tbýt\tVERB\tVB-S---3P-AA---
↪\tMood=Ind|Number=Sing|Person=3|Polarity=Pos|Tense=Pres|VerbForm=Fin|Voice=Act\t0\troot\t_
↪\t_\n2\tzima\tzima\tNOUN\tNNFS1-----A----
↪\tCase=Nom|Gender=Fem|Number=Sing|Polarity=Pos\t1\tnsubj\t_\t_\n3\t.\t.\tPUNCT\tZ:--
↪----------\t_\t1\tpunct\t_\t_\n\n', '# sent_id = 2\n1\tBude\tbýt\tAUX\tVB-S---3F-
↪AA---
↪\tMood=Ind|Number=Sing|Person=3|Polarity=Pos|Tense=Fut|VerbForm=Fin|Voice=Act\t2\taux\t_
↪\t_\n2\tsněžit\tsněžit\tVERB\tVf--------A----
↪\tAspect=Imp|Polarity=Pos|VerbForm=Inf\t0\troot\t_\t_\n3\t.\t.\tPUNCT\tZ:----------
↪--\t_\t2\tpunct\t_\t_\n\n']
```

That's a bit messy, but trust me that `conllu_sents` is just a list of two strings, each string representing one sentence. Or, if you don't trust me:

```
>>> len(conllu_sents)
2
>>> [type(x) for x in conllu_sents]
[<class 'str'>, <class 'str'>]
```

To give you an idea of the format, let's just join the sentences and print them out:

```
>>> print("".join(conllu_sents), end="")
# newdoc
# newpar
# sent_id = 1
1    Je      být     VERB    VB-S---3P-AA---␣
↪Mood=Ind|Number=Sing|Person=3|Polarity=Pos|Tense=Pres|VerbForm=Fin|Voice=Act    0  ␣
↪    root    _       _
2    zima    zima    NOUN    NNFS1-----A----␣
↪Case=Nom|Gender=Fem|Number=Sing|Polarity=Pos    1       nsubj   _       _
3    .       .       PUNCT   Z:-------------- _       1       punct   _       _

# sent_id = 2
1    Bude    být     AUX     VB-S---3F-AA---␣
↪Mood=Ind|Number=Sing|Person=3|Polarity=Pos|Tense=Fut|VerbForm=Fin|Voice=Act     2  ␣
↪    aux     _       _
2    sněžit  sněžit  VERB    Vf--------A---- Aspect=Imp|Polarity=Pos|VerbForm=Inf   ␣
↪0       root    _       _
3    .       .       PUNCT   Z:-------------- _       2       punct   _       _
```

2.1.5 Format conversion

The module can also be used just for loading/dumping data in any of the formats supported by UDPipe. That's what the *load()* and *dump()* functions are for. Input and output formats default to CoNLL-U.

```
>>> from corpy.udpipe import load, dump
>>> sents = list(load(horizontal, "horizontal"))
>>> pprint(sents)
[Sentence(
   comments=['# newdoc', '# newpar', '# sent_id = 1'],
```

(continues on next page)

(continued from previous page)

```
   words=[
     Word(id=0, <root>),
     Word(id=1, form='Je'),
     Word(id=2, form='zima'),
     Word(id=3, form='.')]),
 Sentence(
   comments=['# sent_id = 2'],
   words=[
     Word(id=0, <root>),
     Word(id=1, form='Bude'),
     Word(id=2, form='sněžit'),
     Word(id=3, form='.')])]
>>> print("".join(dump(sents)), end="")
# newdoc
# newpar
# sent_id = 1
1    Je      _       _       _       _       _       _       _       _
2    zima    _       _       _       _       _       _       _       _
3    .       _       _       _       _       _       _       _       _

# sent_id = 2
1    Bude    _       _       _       _       _       _       _       _
2    sněžit  _       _       _       _       _       _       _       _
3    .       _       _       _       _       _       _       _       _
```

You can mix and match this with tagging and parsing the data using a *Model*, if you prefer this more incremental approach:

```
>>> m.tag(sents[0])
>>> m.parse(sents[0])
>>> pprint(sents)
[Sentence(
   comments=['# newdoc', '# newpar', '# sent_id = 1'],
   words=[
     Word(id=0, <root>),
     Word(id=1,
          form='Je',
          lemma='být',
          xpostag='VB-S---3P-AA---',
          upostag='VERB',
          feats=
↪'Mood=Ind|Number=Sing|Person=3|Polarity=Pos|Tense=Pres|VerbForm=Fin|Voice=Act',
          head=0,
          deprel='root'),
     Word(id=2,
          form='zima',
          lemma='zima',
          xpostag='NNFS1-----A----',
          upostag='NOUN',
          feats='Case=Nom|Gender=Fem|Number=Sing|Polarity=Pos',
          head=1,
          deprel='nsubj'),
     Word(id=3,
          form='.',
          lemma='.',
          xpostag='Z:-------------',
```

(continues on next page)

(continued from previous page)

```
          upostag='PUNCT',
          head=1,
          deprel='punct')]),
 Sentence(
   comments=['# sent_id = 2'],
   words=[
     Word(id=0, <root>),
     Word(id=1, form='Bude'),
     Word(id=2, form='sněžit'),
     Word(id=3, form='.')])]
```

As you can see, only the first sentence has been tagged and parsed. Note that the *tag()* and *parse()* methods modify the sentence in place!

2.2 Tokenize and tag text with MorphoDiTa

2.2.1 Overview

The *corpy.morphodita* sub-package offers a more user friendly wrapper around the default Swig-generated Python bindings for the MorphoDiTa morphological tagging and lemmatization framework.

The target audiences are:

- beginner programmers interested in NLP
- seasoned programmers who want to use MorphoDiTa through a more Pythonic interface, without having to dig into the API reference and the examples, and who are not too worried about a possible performance hit as compared with full manual control

Pre-trained tagging models which can be used with MorphoDiTa can be found here. Currently, Czech and English models are available. **Please respect their CC BY-NC-SA 3.0 license!**

At the moment, only a subset of the functionality offered by the MorphoDiTa API is available through *corpy.morphodita* (tokenization, tagging).

If stuck, check out the module's *API reference* for more details.

2.2.2 Tokenization

When instantiating a *Tokenizer*, pass in a string which will determine the type of tokenizer to create. Valid options are `"czech"`, `"english"`, `"generic"` and `"vertical"` (cf. also the `new_*_tokenizer` methods in the MorphoDiTa API reference).

```
>>> from corpy.morphodita import Tokenizer
>>> tokenizer = Tokenizer("generic")
>>> for word in tokenizer.tokenize("foo bar baz"):
...     print(word)
...
foo
bar
baz
```

Alternatively, if you want to use the tokenizer associated with a MorphoDiTa `*.tagger` file you have available, you can instantiate it using *from_tagger()*.

If you're interested in sentence boundaries too, pass `sents=True` to *`tokenize()`*:

```
>>> for sentence in tokenizer.tokenize("foo bar baz", sents=True):
...     print(sentence)
...
['foo', 'bar', 'baz']
```

2.2.3 Tagging

NOTE: Unlike tokenization, tagging in MorphoDiTa requires you to supply your own pre-trained tagging models (see *Overview* above).

Initialize a new tagger:

```
>>> from corpy.morphodita import Tagger
>>> tagger = Tagger("./czech-morfflex-pdt-161115.tagger")
```

Tokenize, tag and lemmatize a text represented as a string:

```
>>> from pprint import pprint
>>> tokens = list(tagger.tag("Je zima. Bude sněžit."))
>>> pprint(tokens)
[Token(word='Je', lemma='být', tag='VB-S---3P-AA---'),
 Token(word='zima', lemma='zima-1', tag='NNFS1-----A----'),
 Token(word='.', lemma='.', tag='Z:-------------'),
 Token(word='Bude', lemma='být', tag='VB-S---3F-AA---'),
 Token(word='sněžit', lemma='sněžit_:T', tag='Vf--------A----'),
 Token(word='.', lemma='.', tag='Z:-------------')]
```

With sentence boundaries:

```
>>> sents = list(tagger.tag("Je zima. Bude sněžit.", sents=True))
>>> pprint(sents)
[[Token(word='Je', lemma='být', tag='VB-S---3P-AA---'),
  Token(word='zima', lemma='zima-1', tag='NNFS1-----A----'),
  Token(word='.', lemma='.', tag='Z:-------------')],
 [Token(word='Bude', lemma='být', tag='VB-S---3F-AA---'),
  Token(word='sněžit', lemma='sněžit_:T', tag='Vf--------A----'),
  Token(word='.', lemma='.', tag='Z:-------------')]]
```

Tag and lemmatize an already sentence-split and tokenized piece of text, represented as an iterable of iterables of strings:

```
>>> tokens = list(tagger.tag([['Je', 'zima', '.'], ['Bude', 'sněžit', '.']]))
>>> pprint(tokens)
[Token(word='Je', lemma='být', tag='VB-S---3P-AA---'),
 Token(word='zima', lemma='zima-1', tag='NNFS1-----A----'),
 Token(word='.', lemma='.', tag='Z:-------------'),
 Token(word='Bude', lemma='být', tag='VB-S---3F-AA---'),
 Token(word='sněžit', lemma='sněžit_:T', tag='Vf--------A----'),
 Token(word='.', lemma='.', tag='Z:-------------')]
```

2.3 Easily generate word clouds

The wordcloud package is great but I find the API a bit ceremonious, especially for beginners. Hence this wrapper to make using it easier.

```
>>> from corpy.vis import wordcloud
>>> import os
>>> wc = wordcloud(os.__doc__)
>>> wc.to_image().show()
```

In a Jupyter notebook, just inspect the `wc` variable to display the wordcloud.

For further details, see the docstring of the *`wordcloud()`* function.

2.4 Rule-based grapheme to phoneme conversion for Czech

In addition to rules, an exception system is also implemented which makes it possible to capture less regular pronunciation patterns.

2.4.1 Usage

The simplest public interface is the *`transcribe()`* function. See its docstring for more information on the types of accepted input as well as on output options and other available customizations. Here are a few usage examples – default output is SAMPA:

```
>>> from corpy.phonetics import cs
>>> cs.transcribe("máš hlad")
[('m', 'a:', 'Z'), ('h\\', 'l', 'a', 't')]
```

But other options including IPA are available:

```
>>> cs.transcribe("máš hlad", alphabet="IPA")
[('m', 'a', ''), ('', 'l', 'a', 't')]
```

Hyphens can be used to prevent interactions between neighboring phones, e.g. assimilation of voicing:

```
>>> cs.transcribe("máš -hlad")
[('m', 'a:', 'S'), ('h\\', 'l', 'a', 't')]
```

As you can see, these special hyphens get deleted in the process of transcription, so if you want a literal hyphen, it must be inside a token with either no alphabetic characters, or at least one other non-alphabetic character:

```
>>> cs.transcribe("- --- -.- -hlad?")
['-', '---', '-.-', '-hlad?']
```

In general, tokens containing non-alphabetic characters (modulo the special treatment of hyphens described above) are passed through as is:

```
>>> cs.transcribe("máš ? hlad")
[('m', 'a:', 'Z'), '?', ('h\\', 'l', 'a', 't')]
```

And you can even configure some of them to constitute a blocking boundary for interactions between phones (notice that unlike in the previous example, "máš" ends with a /S/ → assimilation of voicing wasn't allowed to spread past the ".."):

```
>>> cs.transcribe("máš .. hlad", prosodic_boundary_symbols={".."})
[('m', 'a:', 'S'), '..', ('h\\', 'l', 'a', 't')]
```

Finally, when the input is a single string, it's simply split on whitespace, but you can also provide your own tokenization. E.g. if your input string contains unspaced square brackets to mark overlapping speech, this is probably not the output you want:

```
>>> cs.transcribe("[máš] hlad")
['[máš]', ('h\\', 'l', 'a', 't')]
```

But if you pretokenize the input yourself according to rules that make sense in your situation, you're good to go:

```
>>> cs.transcribe(["[", "máš", "]", "hlad"])
['[', ('m', 'a:', 'Z'), ']', ('h\\', 'l', 'a', 't')]
```

2.4.2 Acknowledgments

The choice of (X-)SAMPA and IPA transcription symbols follows the guidelines published by the Institute of Phonetics, Faculty of Arts, Charles University, Prague, which are hereby gratefully acknowledged.

2.5 Wrangle corpora in the vertical format

2.5.1 Overview

Tools for parsing corpora in the vertical format devised originally for CWB, used also by (No)SketchEngine. It would have been nice if verticals were just standards compliant XML, but they appeared before XML, so they're not. Hence this.

NOTE: The examples below are currently not tested because they require the `syn2015.gz` vertical file to be available, which is large and should not be freely distributed.

```
>>> import pytest
>>> pytest.skip("examples not tested")
```

2.5.2 Iterating over positions in a vertical file

This allows you to iterate over all positions while keeping track of the structural attributes of the structures they're contained within, without risking errors from hand-coding this logic every time you need it.

```
>>> from corpy.vertical import Syn2015Vertical
>>> from pprint import pprint
>>> v = Syn2015Vertical("path/to/syn2015.gz")
>>> for i, position in enumerate(v.positions()):
...     if i % 100 == 0:
...         # structural attributes of position
...         pprint(v.sattrs)
...         print()
...         # position itself
...         pprint(position)
...         print()
...     elif i > 100:
```

(continues on next page)

(continued from previous page)

```
...         break
...
{'doc': {'audience': 'GEN: obecné publikum',
         'author': 'Typlt, Jaromír',
         'authsex': 'M: muž',
         'biblio': 'Typlt, Jaromír (1993): Zápas s rodokmenem. Praha: Pražská '
                   'imaginace.',
         'first_published': '1993',
         'genre': 'X: neuvedeno',
         'genre_group': 'X: neuvedeno',
         'id': 'pi291',
         'isbnissn': '80-7110-132-X',
         'issue': '',
         'medium': 'B: kniha',
         'periodicity': 'NP: neperiodická publikace',
         'publisher': 'Pražská imaginace',
         'pubplace': 'Praha',
         'pubyear': '1993',
         'srclang': 'cs: čeština',
         'subtitle': 'Groteskní mýtus',
         'title': 'Zápas s rodokmenem',
         'translator': 'X',
         'transsex': 'X: neuvedeno',
         'txtype': 'NOV: próza',
         'txtype_group': 'FIC: beletrie'},
'p': {'id': 'pi291:1:1', 'type': 'normal'},
's': {'id': 'pi291:1:1:1'},
'text': {'author': '', 'id': 'pi291:1', 'section': '', 'section_orig': ''}}

Position(word='ZÁPAS', lemma='zápas', tag=UtklTag(pos='N', sub='N', gen='I', num='S',␣
↪case='1', pgen='-', pnum='-', pers='-', tense='-', grad='-', neg='A', act='-', p13=
↪'-', p14='-', var='-', asp='-'), proc='T', afun='ExD', parent='0', eparent='0',␣
↪prep='', p_lemma='', p_tag='', p_afun='', ep_lemma='', ep_tag='', ep_afun='')

{'doc': {'audience': 'GEN: obecné publikum',
         'author': 'Typlt, Jaromír',
         'authsex': 'M: muž',
         'biblio': 'Typlt, Jaromír (1993): Zápas s rodokmenem. Praha: Pražská '
                   'imaginace.',
         'first_published': '1993',
         'genre': 'X: neuvedeno',
         'genre_group': 'X: neuvedeno',
         'id': 'pi291',
         'isbnissn': '80-7110-132-X',
         'issue': '',
         'medium': 'B: kniha',
         'periodicity': 'NP: neperiodická publikace',
         'publisher': 'Pražská imaginace',
         'pubplace': 'Praha',
         'pubyear': '1993',
         'srclang': 'cs: čeština',
         'subtitle': 'Groteskní mýtus',
         'title': 'Zápas s rodokmenem',
         'translator': 'X',
         'transsex': 'X: neuvedeno',
         'txtype': 'NOV: próza',
         'txtype_group': 'FIC: beletrie'},
```

(continues on next page)

(continued from previous page)

```
'p': {'id': 'pi291:1:3', 'type': 'normal'},
's': {'id': 'pi291:1:3:2'},
'text': {'author': '', 'id': 'pi291:1', 'section': '', 'section_orig': ''}}

Position(word='chvil', lemma='chvíle', tag=UtklTag(pos='N', sub='N', gen='F', num='P',
↪ case='2', pgen='-', pnum='-', pers='-', tense='-', grad='-', neg='A', act='-', p13=
↪'-', p14='-', var='-', asp='-'), proc='M', afun='Atr', parent='-1', eparent='-1',␣
↪prep='', p_lemma='několik', p_tag='Ca--4-----------', p_afun='Adv', ep_lemma=
↪'několik', ep_tag='Ca--4-----------', ep_afun='Adv')
```

2.5.3 Performing frequency distribution queries

This can be done elegantly and fairly quickly with *search()*. All you have to do is provide a match function, which identifies positions which the query should match, and a count function, which specifies what should be counted for each match.

The return value is an index of occurrences and the total size of the corpus. The index is a dictionary of numpy array of position indices within the corpus, which can be further processed e.g. using *ipm()* or *arf()* to compute different types of frequencies.

```
>>> from corpy.vertical import Syn2015Vertical, ipm, arf
>>> v = Syn2015Vertical("path/to/syn2015.gz")
# log progress every 50M positions
>>> v.report = 50_000_000
>>> def match(posattrs, sattrs):
...     # match all nouns within txtype_group "FIC: beletrie"
...     return sattrs["doc"]["txtype_group"] == "FIC: beletrie" and posattrs.tag.pos␣
↪== "N"
...
>>> def count(posattrs, sattrs):
...     # at each matched position, record the txtype and lemma
...     return sattrs["doc"]["txtype"], posattrs.lemma
...
>>> index, N = v.search(match, count)
Processed 0 lines in 0:00:00.007382.
Processed 50,000,000 lines in 0:05:58.185566.
Processed 100,000,000 lines in 0:11:35.394294.
```

NOTE: this was run on a desktop workstation, with the data being stored on a networked filesystem. If the performance of any future versions on a similar task becomes significantly worse than this ballpark, it should be considered a bug.

```
# absolute frequency
>>> len(index[("NOV: próza", "plíseň")])
211
# relative frequency (instances per million)
>>> ipm(index[("NOV: próza", "plíseň")], N)
1.747430618598555
# average reduced frequency (takes into account dispersion)
>>> arf(index[("NOV: próza", "plíseň")], N)
54.220727998809153
```

2.5.4 Subclass `Vertical` for your custom corpus

If you have a corpus with a different structure, you can easily adapt the tools by subclassing *`Vertical`*. See its docstring for further info, or the implementation of *`Syn2015Vertical`* for a practical example.

2.6 Command line scripts

CorPy also comes with a few (possibly) handy command line utilities:

- `xc`: Prints frequency information about extended grapheme clusters in text files.
- `zip-verticals`: Zips two verticals of the same corpus with different positional attributes together.

Run them with the `--help` option to get usage instructions.

2.7 `corpy.udpipe`

Tokenizing, tagging and parsing text with UDPipe.

exception `corpy.udpipe.`**UdpipeError**
An error which occurred in the `ufal.udpipe` C extension.

class `corpy.udpipe.`**Model**(*model_path*)
A UDPipe model for tagging and parsing text.

Parameters **model_path** (*`str`*) – Path to the pre-compiled UDPipe model to load.

process(*text*, *, *tag=True*, *parse=True*, *in_format=None*, *out_format=None*)
Process input text, yielding sentences one by one.

The text is always at least tokenized, and optionally morphologically tagged and syntactically parsed, depending on the values of the `tag` and `parse` arguments.

Parameters

- **text** (*`str`*) – Text to process.
- **tag** (*`bool`*) – Perform morphological tagging.
- **parse** (*`bool`*) – Perform syntactic parsing.
- **in_format** (*`None or str`*) – Input format (cf. below for possible values).
- **out_format** (*`None or str`*) – Output format (cf. below for possible values).

The input text is a string in one of the following formats (specified by `in_format`):

- `None`: freeform text, which will be sentence split and tokenized by UDPipe
- `"conllu"`: the CoNLL-U format
- `"horizontal"`: one sentence per line, word forms separated by spaces
- `"vertical"`: one word per line, empty lines denote sentence ends

The output format is specified by `out_format`:

- `None`: native `ufal.udpipe` objects, suitable for further manipulation in Python
- `"conllu"`, `"horizontal"` or `"vertical"`: cf. above
- `"epe"`: the EPE (Extrinsic Parser Evaluation 2017) interchange format

- `"matxin"`: the Matxin XML format
- `"plaintext"`: reconstruct text with original spaces, discarding annotations

New input and output formats may be added with new releases of UDPipe; for an up-to-date list, consult the UDPipe API reference.

tag(*sent*)
Perform morphological tagging on sentence.

Modifies `sent` in place.

Parameters **sent** (*ufal.udpipe.Sentence*) – Sentence to tag.

parse(*sent*)
Perform syntactic parsing on sentence.

Modifies `sent` in place.

Parameters **sent** (*ufal.udpipe.Sentence*) – Sentence to parse.

corpy.udpipe.**load**(*corpus*, *in_format='conllu'*)
Load corpus in input format.

Parameters

- **corpus** (*str*) – The data to load.
- **in_format** (*str*) – Cf. the documentation of *Model.process()*.

Returns A generator of sentences (`ufal.udpipe.Sentence`).

corpy.udpipe.**dump**(*sent_or_sents*, *out_format='conllu'*)
Dump sentence or sentences in output format.

Parameters

- **sent_or_sents** – The data to dump.
- **out_format** (*str*) – Cf. the documentation of *Model.process()*.

Returns A generator of strings, corresponding to the serialized sentences. One final additional string may contain any closing markup, if required by the output format.

corpy.udpipe.**pprint**(*obj*)
Pretty-print object.

This is a convenience wrapper over `IPython.lib.pretty.pprint()` for easier importing.

corpy.udpipe.**pprint_config**(*, *digest=True*)
Configure pretty-printing of `ufal.udpipe` objects.

Parameters **digest** (*bool*) – Show only attributes with interesting values (other than `''` or `-1`)

2.8 corpy.morphodita

Convenient and easy-to-use MorphoDiTa wrappers.

class corpy.morphodita.**Tokenizer**(*tokenizer_type*)
A wrapper API around the tokenizers offered by MorphoDiTa.

Parameters **tokenizer_type** (*str*) – Type of the requested tokenizer (cf. below for possible values).

tokenizer_type is typically one of:

- `"czech"`: a tokenizer tuned for Czech
- `"english"`: a tokenizer tuned for English
- `"generic"`: a generic tokenizer
- `"vertical"`: a simple tokenizer for the vertical format, which is effectively already tokenized (one word per line)

Specifically, the available tokenizers are determined by the `new_*_tokenizer` static methods on the MorphoDiTa `tokenizer` class described in the MorphoDiTa API reference.

static from_tagger(*tagger_path*)
Load tokenizer associated with tagger file.

tokenize(*text*, *sents=False*)
Tokenize *text*.

Parameters

- **text** (*str*) – Text to tokenize.
- **sents** (*bool*) – Whether to signal sentence boundaries by outputting a sequence of lists (sentences).

Returns An iterator over the tokenized text, possibly grouped into sentences if `sents=True`.

Note that MorphoDiTa performs both sentence splitting and tokenization at the same time, but this method iterates over tokens without sentence boundaries by default:

```
>>> from corpy.morphodita import Tokenizer
>>> t = Tokenizer("generic")
>>> for word in t.tokenize("foo bar baz"):
...     print(word)
...
foo
bar
baz
```

If you want to iterate over sentences (lists of tokens), set `sents=True`:

```
>>> for sentence in t.tokenize("foo bar baz", sents=True):
...     print(sentence)
...
['foo', 'bar', 'baz']
```

class corpy.morphodita.**Tagger**(*tagger_path*)
A MorphoDiTa morphological tagger and lemmatizer.

Parameters **tagger_path** (*str*) – Path to the pre-compiled tagging models to load.

tag(*text*, *, *sents=False*, *guesser=False*, *convert=None*)
Perform morphological tagging and lemmatization on text.

If `text` is a string, sentence-split, tokenize and tag that string. If it's an iterable of iterables (typically a list of lists), then take each nested iterable as a separate sentence and tag it, honoring the provided sentence boundaries and tokenization.

Parameters

- **text** (*either str (tokenization is left to the tagger) or iterable of iterables (of str), representing individual sentences*) – Input text.

- **sents** (*bool*) – Whether to signal sentence boundaries by outputting a sequence of lists (sentences).
- **guesser** (*bool*) – Whether to use the morphological guesser provided with the tagger (if available).
- **convert** (*str, one of "pdt_to_conll2009", "strip_lemma_comment" or "strip_lemma_id", or None if no conversion is required*) – Conversion strategy to apply to lemmas and / or tags before outputting them.

Returns An iterator over the tagged text, possibly grouped into sentences if `sents=True`.

```
>>> tagger = Tagger("./czech-morfflex-pdt-161115.tagger")
>>> from pprint import pprint
>>> tokens = list(tagger.tag("Je zima. Bude sněžit."))
>>> pprint(tokens)
[Token(word='Je', lemma='být', tag='VB-S---3P-AA---'),
 Token(word='zima', lemma='zima-1', tag='NNFS1-----A----'),
 Token(word='.', lemma='.', tag='Z:-------------'),
 Token(word='Bude', lemma='být', tag='VB-S---3F-AA---'),
 Token(word='sněžit', lemma='sněžit_:T', tag='Vf--------A----'),
 Token(word='.', lemma='.', tag='Z:-------------')]
>>> tokens = list(tagger.tag([['Je', 'zima', '.'], ['Bude', 'sněžit', '.']]))
>>> pprint(tokens)
[Token(word='Je', lemma='být', tag='VB-S---3P-AA---'),
 Token(word='zima', lemma='zima-1', tag='NNFS1-----A----'),
 Token(word='.', lemma='.', tag='Z:-------------'),
 Token(word='Bude', lemma='být', tag='VB-S---3F-AA---'),
 Token(word='sněžit', lemma='sněžit_:T', tag='Vf--------A----'),
 Token(word='.', lemma='.', tag='Z:-------------')]
>>> sents = list(tagger.tag("Je zima. Bude sněžit.", sents=True))
>>> pprint(sents)
[[Token(word='Je', lemma='být', tag='VB-S---3P-AA---'),
  Token(word='zima', lemma='zima-1', tag='NNFS1-----A----'),
  Token(word='.', lemma='.', tag='Z:-------------')],
 [Token(word='Bude', lemma='být', tag='VB-S---3F-AA---'),
  Token(word='sněžit', lemma='sněžit_:T', tag='Vf--------A----'),
  Token(word='.', lemma='.', tag='Z:-------------')]]
```

tag_untokenized(*text*, *sents=False*, *guesser=False*, *convert=None*)
This is the method `tag()` delegates to when *text* is a string. See docstring for `tag()` for details about parameters.

tag_tokenized(*text*, *sents=False*, *guesser=False*, *convert=None*)
This is the method `tag()` delegates to when *text* is an iterable of iterables of strings. See docstring for `tag()` for details about parameters.

2.9 corpy.vis

Convenience wrappers for visualizing linguistic data.

corpy.vis.**size_in_pixels**(*width*, *height*, *unit='in'*, *ppi=300*)
Convert size in inches/cm to pixels.

Parameters

- **width** – width, measured in *unit*

- **height** – height, measured in *unit*
- **unit** – `"in"` for inches, `"cm"` for centimeters
- **ppi** – pixels per inch

Returns `(width, height)` in pixels

Return type (int, int)

Sample values for ppi:

- for displays: you can detect your monitor's DPI using the following website: <https://www.infobyip.com/detectmonitordpi.php>; a typical value is 96 (of course, double that for HiDPI)
- for print output: 300 at least, 600 is high quality

`corpy.vis.`**`wordcloud`**(*data*, *size=(400, 400)*, ***, *rounded=False*, *fast=True*, *fast_limit=800*, ***kwargs*)

Generate a wordcloud.

If *data* is a string, the wordcloud is generated using the method `WordCloud.generate_from_text()`, which automatically ignores stopwords (customizable with the *stopwords* argument) and includes "collocations" (i.e. bigrams).

If *data* is a sequence or a mapping, the wordcloud is generated using the method `WordCloud.generate_from_frequencies()` and these preprocessing responsibilities fall to the user.

Parameters

- **data** – input data – either one long string of text, or an iterable of tokens, or a mapping of word types to their frequencies; use the second or third option if you want full control over the output
- **size** – size in pixels, as a tuple of integers, (width, height); if you want to specify the size in inches or cm, use the *size_in_pixels()* function to generate this tuple
- **rounded** – whether or not to enclose the wordcloud in an ellipse; incompatible with the *mask* keyword argument
- **fast** – when `True`, optimizes large wordclouds for speed of generation rather than precision of word placement
- **fast_limit** – speed optimizations for "large" wordclouds are applied when the requested canvas size is larger than `fast_limit**2`
- **kwargs** – remaining keyword arguments are passed on to the `wordcloud.WordCloud` initializer

Returns The word cloud.

Return type `wordcloud.WordCloud`

2.10 corpy.phonetics.cs

Perform rule-based phonetic transcription of Czech.

Some frequent exceptions to the otherwise fairly regular orthography-to-phonetics mapping are overridden using a pronunciation lexicon.

class `corpy.phonetics.cs.`**`Phone`**(*value: str*, ***, *word_boundary: bool = False*)

A single phone.

You probably don't need to create these by hand, but they will be returned to you from *transcribe()*.

class corpy.phonetics.cs.**ProsodicUnit**(*orthographic: List[str]*)
A prosodic unit which should be transcribed as a whole.

This means that various connected speech processes are emulated at word boundaries within the unit as well as within words.

Parameters **orthographic** (*list of str*) – The orthographic transcript of the prosodic unit.

phonetic(*, *alphabet: str = 'SAMPA'*, *hiatus=False*) → List[Tuple[str, ...]]
Phonetic transcription of ProsodicUnit.

corpy.phonetics.cs.**transcribe**(*phrase: Union[str, Iterable[str]], *, alphabet='sampa', hiatus=False, prosodic_boundary_symbols={}*) → List[Union[str, Tuple[str, ...]]]
Phonetically transcribe *phrase*.

phrase is either a string (in which case it is split on whitespace) or an iterable of strings (in which case it's considered as already tokenized by the user).

Transcription is attempted for tokens which consist purely of alphabetical characters and possibly hyphens (`-`). Other tokens are passed through unchanged. Hyphens have a special role: they prevent interactions between graphemes or phones from taking place, which means you can e.g. cancel assimilation of voicing in a cluster like `tb` by inserting a hyphen between the graphemes: `t-b`. They are removed from the final output. If you want a **literal hyphen**, it must be inside a token with either no alphabetic characters, or at least one other non-alphabetic character (e.g. `-`, `---`, `-hlad?`, etc.).

Returns a list where **transcribed tokens** are represented as **tuples of strings** (phones) and **non-transcribed tokens** (which were just passed through as-is) as plain **strings**.

alphabet is one of SAMPA, IPA, CS or CNC (case insensitive) and determines the symbol alphabet used in the phonetic transcript.

When `hiatus=True`, a /j/ phone is added between a high front vowel and a subsequent vowel.

Various connected speech processes such as assimilation of voicing are emulated even across word boundaries. By default, this happens **irrespective of intervening non-transcribed tokens**. If you want some types of non-transcribed tokens to constitute an obstacle to interactions between phones, pass them as a set via the *prosodic_boundary_symbols* argument. E.g. `prosodic_boundary_symbols={"?", ".."}` will prevent CSPs from being emulated across `?` and `..` tokens.

2.11 corpy.vertical

Parse and query corpora in the vertical format.

class corpy.vertical.**Vertical**(*path*)
Base class for a corpus in the vertical format.

Create subclasses for specific corpora by at least specifying a list of *struct_names* and *posattrs* as class attributes.

Parameters **path** (*str*) – Path to the vertical file to work with.

struct_names = []
A list of expected structural attribute tag names.

posattrs = []
A list of expected positional attributes.

open()
Open the vertical file in `self.path`.

Override this method in subclasses to specify alternative ways of opening, e.g. using `gzip.open()`.

parse_position(*position*)
Parse a single position from the vertical.

Override this method in subclasses to hook into the position parsing process.

positions(*parse_sattrs=True*, *ignore_fn=None*, *hook_fn=None*)
Iterate over the positions in the vertical.

At any point during the iteration, the structural attributes corresponding to the current position are accessible via `self.sattrs`.

Parameters

- **parse_sattrs** (*bool*) – Whether to parse structural attrs into a dict (default) or just leave the original string (faster).
- **ignore_fn** (*function(posattrs, sattrs)*) – If given, then evaluated at each position; if it returns `True`, then the position is completely ignored.
- **hook_fn** (*function(posattrs, sattrs)*) – If given, then evaluated at each position.

search(*match_fn*, *count_fn=None*, ***kwargs*)
Search the vertical, creating an index of what's been found.

Parameters

- **match_fn** (*function(match_fn, count_fn)*) – Evaluated at each position to see if the position matches the given search.
- **count_fn** – Evaluated at each **matching** position to determine what should be counted at that position (in the sense of being tallied as part of the resulting frequency distribution). If it returns a list, it's understood as a list of things to count.
- **kwargs** – Passed on to *positions()*.

Returns The frequency index of counted "things" and the size of the corpus.

Return type (dict, int)

class corpy.vertical.**Syn2015Vertical**(*path*)
A subclass of *Vertical* for the SYN2015 corpus.

Refer to *Vertical* for API details.

open()
Open the vertical file in `self.path`.

Override this method in subclasses to specify alternative ways of opening, e.g. using `gzip.open()`.

parse_position(*position*)
Parse a single position from the vertical.

Override this method in subclasses to hook into the position parsing process.

corpy.vertical.**ipm**(*occurrences*, *N*)
Relative frequency of *occurrences* in corpus, in instances per million.

corpy.vertical.**arf**(*occurrences*, *N*)
Average reduced frequency of *occurrences* in corpus.

class corpy.vertical.**ShuffledSyn2015Vertical**(*path*)
A subclass of *Vertical* for the SYN2015 corpus, shuffled.

Refer to *Vertical* for API details.

2.12 corpy.util

Small utility functions.

corpy.util.**head**(*collection*, *first_n=None*)
Inspect *collection*, truncated if too long.

If first_n=None, an appropriate value is determined based on the type of the collection.

corpy.util.**cmp**(*lhs*, *rhs*, *test='__eq__'*)
Wrap assert statement to automatically raise an informative error.

CHAPTER

THREE

LICENSE

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX