Getting Started with LatinCy

Patrick J. Burns

2024-08-26

Table of contents

Preface

"Getting Started with LatinCy" is an always-a-work-in-progress combination of documentation and demo notebooks for working with the LatinCy models on a variety of Latin text analysis and NLP tasks.

Key links

Models: https://huggingface.co/latincy
Universe: https://spacy.io/universe/project/latincy
Preprint: https://arxiv.org/abs/2305.04365

This book has been written using Jupyter notebooks which have then been collated with Quarto. To learn more about Quarto books visit https://quarto.org/docs/books.

Acknowledgements

Thank you to all of the readers who have submitted corrections via GitHub PRs: @sjhuskey.

Abbreviations

Where possible, I will include references to standard NLP works using the following abbreviations:

NLTK Bird, S., Klein, E., and Loper, E. 2015. *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. 2nd Edition. https://www.nltk.org/book/. (Bird, Klein, and Loper 2015)
SLP Jurafsky, D., and Martin, J.H. 2020. *Speech and Language Processing*. 3rd Edition, Draft. https://web.stanford.edu/~jurafsky/slp3/. (Jurafsky and Martin 2020)

Part I

Front Matter

Part II

Model/Pipeline Basics

1 Installing LatinCy models

1.1 Installing spaCy

The LatinCy models are designed to work with the spaCy natural language platform, so you will need to have this package installed before anything else. The following cell has the pip install command for spaCy. At the time of writing, the latest version available for spaCy compatible with LatinCy is v3.7.5.

NB: To run the cells below, uncomment the commands by removing the # at the beginning of the line. The exclamation point at the beginning of the line is shorthand for the `%system` magic command in Jupyter which can be used to run shell commands from within a notebook.

```
# !pip install -U spacy
```

1.2 Installing the LatinCy models

LatinCy models are currently available in three sizes: 'sm', 'md', and 'lg'. We will use the different models throughout the tutorials, so let's install all three now so that they are available for future chapters.

```
# ! pip install "la-core-web-sm @ https://huggingface.co/latincy/la_core_web_sm/resolve/main/
# ! pip install "la-core-web-md @ https://huggingface.co/latincy/la_core_web_md/resolve/main/
# ! pip install "la-core-web-lg @ https://huggingface.co/latincy/la_core_web_lg/resolve/main/
# ! pip install "la-core-web-trf @ https://huggingface.co/latincy/la_core_web_trf/resolve/ma
```

1.3 Installing additional packages

We will also use other packages throughout these tutorials. They are included here for your convenience, as well as in the `requirements.txt` file in the code repository for this Quarto book.

```
# !pip install pandas
# !pip install matplotlib
# !pip install seaborn
# !pip install scikit-learn
# !pip install tqdm
```

2 Loading LatinCy models

2.1 Loading LatinCy models with spaCy

```
# Imports

import spacy
from pprint import pprint
```

```
nlp = spacy.load('la_core_web_lg')
```

2.2 Creating a spaCy doc from a string

```
text = "Haec narrantur a poetis de Perseo."
doc = nlp(text)
print(doc)
```

```
Haec narrantur a poetis de Perseo.
```

```
print(type(doc))
```

```
<class 'spacy.tokens.doc.Doc'>
```

```
print(doc.__repr__())
```

```
Haec narrantur a poetis de Perseo.
```

```
print(doc.text)
```

```
Haec narrantur a poetis de Perseo.
```

```
print(type(doc.text))
```

```
<class 'str'>
```

3 LatinCy components and their annotations

3.1 LatinCy components and their annotations

```
# Imports & setup

import spacy
import pandas as pd
from pprint import pprint
nlp = spacy.load('la_core_web_lg')
text = "avus eius Acrisius appellabatur."
doc = nlp(text)
print(doc)
```

```
avus eius Acrisius appellabatur.
```

Here are the components provided by the default LatinCy models…

```
pprint(nlp.pipe_names)
```

```
['senter',
 'normer',
 'tok2vec',
 'tagger',
 'morphologizer',
 'trainable_lemmatizer',
 'parser',
 'lookup_lemmatizer',
 'ner']
```

The dataframe below summarizes the key annotations provided by these components…

```
data = []

for token in doc:
    data.append([token.text, token.norm_, token.lemma_, token.pos_, token.tag_, token.morph.t

df = pd.DataFrame(data, columns=['text', 'norm', 'lemma', 'pos', 'tag', 'morph', 'dep', 'ent_

df
```

	text	norm	lemma	pos	tag	morph
0	avus	auus	auus	NOUN	noun	Case=Nom|Gender=Masc|Number=Sin
1	eius	eius	is	PRON	pronoun	Case=Gen|Gender=Masc|Number=Sing
2	Acrisius	acrisius	Acrisius	PROPN	proper_noun	Case=Nom|Gender=Masc|Number=Sin
3	appellabatur	appellabatur	appello	VERB	verb	Mood=Ind|Number=Sing|Person=3|Ten
4	.	.	.	PUNCT	punc	

Part III

NLP Tasks

4 Sentence segmentation

4.1 Sentence segmentation with LatinCy

Sentence segmentation is the task of splitting a text into sentences. For the LatinCy models, this is a task been trained using spaCy's `senter` factory to terminate sentences at both strong and weak stops, following the example of Clayman (1981) (see also, Wake (1957), Janson (1964)), who writes: "If all stops are made equivalent, i.e. if no distinction is made between the strong stop, weak stop and interrogation mark, editorial differences will be kept to a minimum."

Given a spaCy `Doc`, the `sents` attribute will produce a `generator` object with the sentence from that document as determined by the dependency parser. Each sentence is a `Span` object with the start and end token indices from the original `Doc`.

```
# Imports & setup

import spacy
from pprint import pprint
nlp = spacy.load('la_core_web_lg')
text = "Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eiu
doc = nlp(text)
print(doc)
```

```
Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acrisi
```

```
sents = doc.sents
print(type(sents))
```

```
<class 'generator'>
```

Like all `Span` objects, the text from each sentence can be retrieved with the `text` attribute. For convenience below, we convert the generator to list so that we can iterate over it multiple times. Here are the three (3) sentences identified in the example text as well as an indication of the sentences' `type`, i.e. `<class 'spacy.tokens.span.Span'>`.

```
sents = list(sents)

for i, sent in enumerate(sents, 1):
    print(f'{i}: {sent.text}')
```

```
1: Haec narrantur a poetis de Perseo.
2: Perseus filius erat Iovis, maximi deorum.
3: Avus eius Acrisius appellabatur.
```

```
sent = sents[0]
print(type(sent))
```

```
<class 'spacy.tokens.span.Span'>
```

Sentences have the same atrributes/methods available to them as any span (listed in the next cell). Following are some attibutes/methods that may be particularly relevant to working with sentences.

```
sent_methods = [item for item in dir(sent) if '_' not in item]
pprint(sent_methods)
```

```
['conjuncts',
 'doc',
 'end',
 'ents',
 'id',
 'label',
 'lefts',
 'rights',
 'root',
 'sent',
 'sentiment',
 'sents',
 'similarity',
 'start',
 'subtree',
 'tensor',
 'text',
 'vector',
 'vocab']
```

You can identify the `root` of the sentence as determined by the dependency parser. Assuming the parsing in correct, this will be the main verb of the sentence.

```
print(sent.root)
```

```
narrantur
```

Each word in the sentence has an associated vector. Sentence (any `Span` in fact) has an associated vector as well that is the mean of the vectors of the words in the sentence. As this example uses the `lg` model, the vector has a length of 300.

```
print(sent.vector.shape)
```

```
(300,)
```

This vector then can be used to compute the similarity between two sentences. Here we see our example sentence compared to two related sentence: 1. a sentence where the character referred to is changed from Perseus to Ulysses; and 2. the active-verb version of the sentence.

```
sent.similarity(nlp('Haec narrantur a poetis de Ulixe.'))
```

```
0.9814933448498585
```

```
sent.similarity(nlp('Haec narrant poetae de Perseo.'))
```

```
0.7961655550941479
```

We can retrieve the start and end indices from the original document for each sentence.

```
sent_2 = sents[1]
start = sent_2.start
end = sent_2.end
print(start)
print(end)
print(sent_2.text)
print(doc[start:end].text)
```

```
7
15
Perseus filius erat Iovis, maximi deorum.
Perseus filius erat Iovis, maximi deorum.
```

References

SLP Chapter 2, Section 2.4.5 “Sentence Segmentation”, pp. 24 [link] (https://web.stanford.edu/~jurafsky/slp3/)
spaCy SentenceRecognizer

5 Word Tokenization

5.1 Word tokenization with LatinCy

```
# Imports & setup

import spacy
from pprint import pprint
nlp = spacy.load('la_core_web_md')
text = "Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus ei
doc = nlp(text)
print(doc)
```

```
Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acris
```

Word tokenization is the task of splitting a text into words (and wordlike units like punctuation, numbers, etc.). For the LatinCy models, tokenization is the fundamental pipeline component on which all other components depend. SpaCy uses non-destructive, "canonical" tokenization, i.e. non-destructive, in that the original text can be untokenized, so to speak, based on `Token` annotations and canonical in that indices are assigned to each token during this process and these indices are used to refer to the tokens in other annotations. (Tokens *can* be separated or merged, but this requires the user to actively undo and redefine the tokenization output.) LatinCy uses a modified version of the default spaCy tokenizer that recognizes and splits enlitic *-que* using a rules-based process. (NB: It is in the LatinCy development plan to move enclitic splitting to a separate post-tokenization component.)

The spaCy `Doc` object is an iterable and tokens are the iteration unit.

```
tokens = [item for item in doc]
print(tokens)
```

```
[Haec, narrantur, a, poetis, de, Perseo, ., Perseus, filius, erat, Iovis, ,, maximi, deorum,
```

```
token = tokens[0]
print(type(token))
```

```
<class 'spacy.tokens.token.Token'>
```

The text content of a `Token` object can be retrieved with the `text` attribute.

```
for i, token in enumerate(tokens, 1):
    print(f'{i}: {token.text}')
```

```
1: Haec
2: narrantur
3: a
4: poetis
5: de
6: Perseo
7: .
8: Perseus
9: filius
10: erat
11: Iovis
12: ,
13: maximi
14: deorum
15: .
16: Avus
17: eius
18: Acrisius
19: appellabatur
20: .
```

Note again that the token itself is a spaCy `Token` object and that the `text` attribute returns a Python string even though their representations in the Jupyter Notebook look the same.

```
token = tokens[0]
print(f'{type(token)} -> {token}')
print(f'{type(token.text)} -> {token.text}')
```

```
<class 'spacy.tokens.token.Token'> -> Haec
<class 'str'> -> Haec
```

5.1.1 Token attributes and methods related to tokenization

Here are some atrributes/methods available for spaCy `Token` objects that are relevant to word tokenization.

SpaCy keeps track of both the token indices and the character offsets within a doc using either the `i` or `idx` attributes, respectively…

```
print(token.doc)
```

```
Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acris
```

```
# token indices

for token in doc:
    print(f'{token.i}: {token.text}')
```

```
0: Haec
1: narrantur
2: a
3: poetis
4: de
5: Perseo
6: .
7: Perseus
8: filius
9: erat
10: Iovis
11: ,
12: maximi
13: deorum
14: .
15: Avus
16: eius
17: Acrisius
18: appellabatur
19: .
```

This is functionally equivalent to using `enumerate`…

```
# token indices, with enumerate

for i, token in enumerate(doc):
    print(f'{i}: {token.text}')
```

```
0: Haec
1: narrantur
2: a
3: poetis
4: de
5: Perseo
6: .
7: Perseus
8: filius
9: erat
10: Iovis
11: ,
12: maximi
13: deorum
14: .
15: Avus
16: eius
17: Acrisius
18: appellabatur
19: .
```

Another indexing option is the `idx` attribute which is the character offset of the token in the original `Doc` object.

```
# character offsets,
for token in doc:
    print(f'{token.idx}: {token.text}')
```

```
0: Haec
5: narrantur
15: a
17: poetis
24: de
27: Perseo
33: .
35: Perseus
```

```
43: filius
50: erat
55: Iovis
60: ,
62: maximi
69: deorum
75: .
77: Avus
82: eius
87: Acrisius
96: appellabatur
108: .
```

Observe these `idx` attributes relate to the character offsets from the original `Doc`. To illustrate the point, we will replace spaces with an underscore in the output. We can see from the output above that *narrantur* begins at `idx` 5 and that the next word *a* begins at `idx` 15. Yet *narrantur* is only 9 characters long and the difference between these two numbers is 10! This is because we need to account for whitespace in the original `Doc`. This is handled by the attribute `text_with_ws`.

```
print(doc.text[5:15].replace(' ', '_'))
```

```
narrantur_
```

```
print(f'text -> {doc[1].text} (length {len(doc[1].text)})')
print()
print(f'text_with_ws -> {doc[1].text_with_ws} (length {len(doc[1].text_with_ws)})')
```

```
text -> narrantur (length 9)

text_with_ws -> narrantur  (length 10)
```

Accordingly, using the `text_with_ws` attribute (as opposed to simply the `text` attribute) we can reconstruct the original text. This is what was meant above by "non-destructive" tokenization. Look at the difference between a text joined using the `text` attribute and one joined using the `text_with_ws` attribute.

```
joined_tokens = ' '.join([token.text for token in doc])
print(joined_tokens)
print(joined_tokens == doc.text)

print()

reconstructed_text = ''.join([token.text_with_ws for token in doc])
print(reconstructed_text)
print(reconstructed_text == doc.text)
```

```
Haec narrantur a poetis de Perseo . Perseus filius erat Iovis , maximi deorum . Avus eius Ac
False

Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acris
True
```

Because spaCy tokenization is set from the outset, you can traverse the tokens in a `Doc` objects from the tokens themselves using the `nbor` method. This method takes an integer argument that specifies the number of tokens to traverse. A positive integer traverses the tokens to the right, a negative integer traverses the tokens to the left.

```
print(doc[:6])
print('-----')
print(f'{doc[3]}, i.e. i = 3')
print(f'{doc[3].nbor(-1)}, i.e. i - 1 = 2')
print(f'{doc[3].nbor(-2)}, i.e. i - 2 = 1')
print(f'{doc[3].nbor(1)}, i.e. i + 1 = 4')
print(f'{doc[3].nbor(2)}, i.e. i + 2 = 5')
```

```
Haec narrantur a poetis de Perseo
-----
poetis, i.e. i = 3
a, i.e. i - 1 = 2
narrantur, i.e. i - 2 = 1
de, i.e. i + 1 = 4
Perseo, i.e. i + 2 = 5
```

5.1.2 Customization of the spaCy tokenizer in LatinCy

LatinCy aims to tokenize the *que* enclitic in Latin texts. As noted above this is currently done through a rule-based approach. Here is the custom tokenizer code (beginning at this line in

the code) followed by a description of the process. Note that this process is based on the following recommendations in the spaCy documentation: https://spacy.io/usage/training#custom-tokenizer.

```
from spacy.util import registry, compile_suffix_regex

@registry.callbacks("customize_tokenizer")
def make_customize_tokenizer():
    def customize_tokenizer(nlp):
        suffixes = nlp.Defaults.suffixes + [
            "que",
            "qve",
        ]
        suffix_regex = compile_suffix_regex(suffixes)
        nlp.tokenizer.suffix_search = suffix_regex.search

        for item in que_exceptions:
            nlp.tokenizer.add_special_case(item, [{"ORTH": item}])
            nlp.tokenizer.add_special_case(item.lower(), [{"ORTH": item.lower()}])
            nlp.tokenizer.add_special_case(item.title(), [{"ORTH": item.title()}])
            nlp.tokenizer.add_special_case(item.upper(), [{"ORTH": item.upper()}])

    return customize_tokenizer
```

Basically, we treat *que* (and its case and u/v norm variants) as punctuation. These are added to the `Defaults.suffixes`. If no other intervention were made, then any word ending in *que* or a variant would be split into a before-*que* part and *que*. Since there are large number of relatively predictable words that end in *que*, these are maintained in a list called `que_exceptions`. All of the words in the `que_exceptions` list are added as a "special case" using the tokenizer's `add_special_case` method and so will not be split. The `que_exceptions` lists is as follows:

```
que_exceptions = ['quisque', 'quidque', 'quicque', 'quodque', 'cuiusque', 'cuique', 'quemque
```

You can see these words in the `rules` attribute of the tokenizer.

```
# Sample of 10 que rules from the custom tokenizer

tokenizer_rules = nlp.tokenizer.rules
print(sorted(list(set([rule.lower() for rule in tokenizer_rules if 'que' in rule])))[:10])
```

```
['absque', 'abusque', 'adaeque', 'adusque', 'aeque', 'antique', 'atque', 'circumundique', 'co
```

With the exception of basic enclitic splitting, the LatinCy tokenizer is the same as the default spaCy tokenizer. The default spaCy tokenizer is described in detail in the spaCy documentation. Here are some useful attributes/methods for working with LatinCy.

Tokenize a string without any other pipeline annotations with a call.

```
tokens = nlp.tokenizer(text)
print(tokens)
print(tokens[0].text)
print(tokens[0].lemma_) # Note that there is no annotation here because since the tokenizer
```

```
Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acris
Haec
```

A list of texts can be tokenized in one pass with the `pipe` method. This yields a generator object where each item is `Doc` object of tokenized-only texts

```
texts = ["Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum; avus
tokens = list(nlp.tokenizer.pipe(texts))

print(len(tokens)) # number of documents
print(len(tokens[0])) # number of tokens in first document
```

```
3
76
```

You can get an explanation of the tokenization "decisions" using the `explain` method. In the example below, we see how the *que* in *virumque* is treated as a suffix (as discussed above) and so is split during tokenization.

```
tok_exp = nlp.tokenizer.explain('arma virumque cano')
print(tok_exp)
```

```
[('TOKEN', 'arma'), ('TOKEN', 'virum'), ('SUFFIX', 'que'), ('TOKEN', 'cano')]
```

```
tokens = nlp.tokenizer('arma uirumque cano')
for i, token in enumerate(tokens):
    print(f'{i}: {token.text}')
```

```
0: arma
1: uirum
2: que
3: cano
```

References

SLP Chapter 2, Section 2.4.2 “Word Tokenization”, pp. 18-20 link

6 Lemmatization

6.1 Lemmatization with LatinCy

Lemmatization is the task of mapping a token in a text to its dictionary headword. With the default LatinCy pipelines, two components are used to perform this task: 1. spaCy's `Edit Tree Lemmatizer` and 2. a second custom `Lookup Lemmatizer`, named in the pipeline "trainable_lemmatizer" and "lookup_lemmatizer" respectively. In the first lemmatization pass, a probabilistic tree model is used to predict the transformation from the token form to its lemma. A second pass is made at the end of the pipeline which checks the token form against a large (~1M item) lemma dictionary (i.e. lookups) for ostensibly unambiguous forms; if a match is found, the lemma is overwritten with the corresponding value from lookup. The two-pass logic largely follows the approach recommended in Burns (2018) and Burns (2020), as facilitated by the spaCy pipeline architecture.

```
# Imports & setup

import spacy
from pprint import pprint
nlp = spacy.load('la_core_web_sm')
text = "Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus ei
doc = nlp(text)
print(doc)
```

```
Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acris
```

Note here the two lemmatizer components that are included in the pipeline, i.e. "trainable_lemmatizer" and "lookup_lemmatizer"...

```
print(nlp.pipe_names)
```

```
['senter', 'normer', 'tok2vec', 'tagger', 'morphologizer', 'trainable_lemmatizer', 'parser',
```

Once a text is annotated using the LatinCy pipeline, i.e. as part of the `Doc` creation process, lemmas can be found as annotations of the `Token` objects…

```
sample_token = doc[0]

print(f'Sample token: {sample_token.text}')
print(f'Sample lemma: {sample_token.lemma_}')
```

```
Sample token: Haec
Sample lemma: hic
```

```
import tabulate

data = []

tokens = [item for item in doc]

for token in tokens:
    data.append([token.text, token.lemma_])

print(tabulate.tabulate(data, headers=['Text', 'Lemma']))
```

```
Text          Lemma
------------  --------
Haec          hic
narrantur     narro
a             ab
poetis        poeta
de            de
Perseo        Perseo
.             .
Perseus       Perseus
filius        filius
erat          sum
Iovis         Iuppiter
,             ,
maximi
deorum        deus
.             .
Avus          auus
eius          is
```

```
Acrisius      Acrisius
appellabatur  appello
.             .
```

The `lemma_` attribute has the type `str` and so is compatible with all string operations…

```
print(f'Token: {tokens[0].text}')
print(f'Lemma: {tokens[0].lemma_}')
print(f'Lowercase lemma: {tokens[0].lemma_.lower()}')
```

```
Token: Haec
Lemma: hic
Lowercase lemma: hic
```

The `lemma_` attribute, though, is only the human-readable version of the lemma. Internally, spaCy uses a hash value to represent the lemma, which is stored in the `lemma` attribute. (Note the lack of trailing underscore.)

```
print(f'Token: {tokens[1].text}')
print(f'Human-readable lemma: {tokens[1].lemma_}')
print(f'spaCy lemma key: {tokens[1].lemma}')
```

```
Token: narrantur
Human-readable lemma: narro
spaCy lemma key: 11361982710182407617
```

In order to compare the two different lemmatization passes, we can create two copies of the LatinCy pipeline, each with one of the two lemmatizers removed…

```
import copy

P1 = copy.deepcopy(nlp)
P1.disable_pipes(['tagger', 'morphologizer', 'lookup_lemmatizer'])
print(f'First pipeline components: {P1.pipe_names}')

P2 = copy.deepcopy(nlp)
P2.disable_pipes(['tagger', 'morphologizer', 'trainable_lemmatizer'])
print(f'Second pipeline components: {P2.pipe_names}')
```

```
First pipeline components: ['senter', 'normer', 'tok2vec', 'trainable_lemmatizer', 'parser',
Second pipeline components: ['senter', 'normer', 'tok2vec', 'parser', 'lookup_lemmatizer', '
```

We can then run the same text through both pipelines and compare the results side-by-side…

```
P1_annotations = P1(text)
P2_annotations = P2(text)

data = []

for p1_token, p2_token in zip(P1_annotations, P2_annotations):
    data.append([p1_token.text, p1_token.lemma_, p2_token.lemma_])

print(tabulate.tabulate(data, headers=['Text', 'Trainable lemmatizer', 'Lookup lemmatizer']))
```

```
Text          Trainable lemmatizer    Lookup lemmatizer
------------  ----------------------  -------------------
Haec          hic
narrantur     narro                   narro
a             ab
poetis        poetus                  poeta
de            de
Perseo        Perseo
.             .                       .
Perseus       Perseus
filius        filius                  filius
erat          sum                     sum
Iovis         Iovis                   Iuppiter
,             ,                       ,
maximi
deorum        deus                    deus
.             .                       .
Avus          avus                    auus
eius          is                      is
Acrisius      Acrisius
appellabatur  appello                 appello
.             .                       .
```

Note specifically the lemmatization of *Iovis*—since it is a highly irregular form, it is not surprising that the Edit Tree Lemmatizer has manufactured a potential, but incorrect, lemma based on the root *Iov-*. Since *Iovis* is an unambiguous Latin form and has been added to the LatinCy lookups, the Lookup Lemmatizer steps in to correct the erroneous first pass. The lookup data can be found a custom form of the spaCy `spacy-lookups-data` package [https://github.com/diyclassics/spacy-lookups-data/tree/master/spacy_lookups_data/data]. These lookups are installed as a dependency of each of the LatinCy pipelines.

The code below shows how to access the lookup data directly…

```
# Load the lookups data

from spacy.lookups import Lookups, load_lookups

blank_nlp = spacy.blank("la")
lookups = Lookups()

lookups_data = load_lookups(lang=blank_nlp.vocab.lang, tables=["lemma_lookup"])
LOOKUPS = lookups_data.get_table("lemma_lookup")

print(LOOKUPS['Iovis'])
```

```
Iuppiter
```

References

NLTK Chapter 3, Section 3.6 “Normalizing text” link
SLP Chapter 2, Section 2.6 “Word normalization, lemmatization and stemming”, pp. 23-24 link
spaCy EditTreeLemmatizer link

7 POS Tagging

7.1 POS Tagging with LatinCy

Part-of-speech tagging is the task of mapping a token in a text to its part of speech, whether 'noun' or 'verb' or 'preposition' and so on. There are two components in the default LatinCy pipelines that provide such annotations, that is the `tagger` and the `morphologizer`. Ostensibly, the `tagger` provides language-specific, fine-grain POS tags and the `morphologizer` provides coarse-grain tags (as defined by the UD Universal POS tags); at present, the LatinCy models have a high degree of overlap between these two tagsets and there are effectively *no* fine-grain tags.

```
# Imports & setup

import spacy
from pprint import pprint
nlp = spacy.load('la_core_web_lg')
text = "Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus ei
doc = nlp(text)
print(doc)
```

```
Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acris
```

Note here the two "tagging" components that are included in the pipeline, i.e. "tagger" and "morphologizer"...

```
print(nlp.pipe_names)
```

```
['senter', 'normer', 'tok2vec', 'tagger', 'morphologizer', 'trainable_lemmatizer', 'parser',
```

Once a text is annotated using the LatinCy pipeline, i.e. as part of the `Doc` creation process, tags can be found as annotations of the `Token` objects. The coarse-grain tags are stored in the `pos_` attribute; fine-grain tags are stored in the `tag_` attribute.

```
sample_token = doc[1]

print(f'Sample token: {sample_token.text}')
print(f'Sample POS: {sample_token.pos_}')
print(f'Sample TAG: {sample_token.tag_}')
```

```
Sample token: narrantur
Sample POS: VERB
Sample TAG: verb
```

Note the high degree of overlap between the coarse-grain and fine-grain tags in the LatinCy models in the chart below. That said, it is perhaps worth paying more attention to where the tagsets do not overlap. In the LatinCy training, conventions for classes of words to be labeled, say, "DET" (as in the *Haec* `pos_`) or "AUX" (as in *est*) are inferred from usage in the six different treebanks used. A sense of the inconsistency in the tagsets can be gleaned from the following page: (https://universaldependencies.org/la/); note also the important work of Gamba and Zeman (2023) in this area.

```
import tabulate

data = []

tokens = [item for item in doc]

for token in tokens:
    data.append([token.text, token.pos_, token.tag_])

print(tabulate.tabulate(data, headers=['Text', "POS", "TAG"]))
```

```
Text          POS    TAG
------------  -----  -----------
Haec          DET    pronoun
narrantur     VERB   verb
a             ADP    preposition
poetis        NOUN   noun
de            ADP    preposition
Perseo        PROPN  proper_noun
.             PUNCT  punc
Perseus       PROPN  proper_noun
filius        NOUN   noun
erat          AUX    verb
```

```
Iovis          PROPN  proper_noun
,              PUNCT  punc
maximi         ADJ    adjective
deorum         NOUN   noun
.              PUNCT  punc
Avus           NOUN   noun
eius           PRON   pronoun
Acrisius       PROPN  proper_noun
appellabatur   VERB   verb
.              PUNCT  punc
```

As with the lemma annotations, the `pos_` and `tag_` attributes are only the human-readable of the lemma. Internally, spaCy uses a hash value to represent this, again noting the lack of trailing underscore…

```
print(f"Token: {tokens[1].text}")
print(f'Human-readable TAG: {tokens[1].tag_}')
print(f'spaCy TAG key: {tokens[1].tag}')
```

```
Token: narrantur
Human-readable TAG: verb
spaCy TAG key: 6360137228241296794
```

This tag 'key' can be looked up in spaCy's `NLP.vocab.strings` atrribute…

```
T = nlp.get_pipe('tagger')
tag_lookup = T.vocab.strings[6360137228241296794] # also would work on `nlp`, i.e. nlp.vocab

print(f'TAG key: {tokens[1].tag}')
print(f'Human-readable TAG: {tag_lookup}')
```

```
TAG key: 6360137228241296794
Human-readable TAG: verb
```

The same process applies to the POS 'keys'…

```
print(f'Token: {tokens[1].text}')
print(f'Human-readable POS: {tokens[1].pos_}')
print(f'spaCy POS key: {tokens[1].pos}')
```

```
Token: narrantur
Human-readable POS: VERB
spaCy POS key: 100
```

```
M = nlp.get_pipe("morphologizer")
pos_lookup = M.vocab.strings[100]
print(f'POS key: {tokens[1].pos}')
print(f'Human-readable POS: {pos_lookup}')
```

```
POS key: 100
Human-readable POS: VERB
```

We can use the `label_data` attribute from the `morphologizer` component to derive the complete (coarse-grain) tagset…

```
def split_pos(morph):
    if 'POS=' in morph:
        return morph.split('POS=')[1].split('|')[0]
    else:
        return None

tagset = sorted(list(set([split_pos(k) for k, v in M.label_data['morph'].items() if split_pos
print(tagset)
```

```
['ADJ', 'ADP', 'ADV', 'AUX', 'CCONJ', 'DET', 'INTJ', 'NOUN', 'NUM', 'PART', 'PRON', 'PROPN',
```

SpaCy has an `explain` methods that can show human-readable descriptions of these standard tags…

```
data = []

for tag in tagset:
    data.append([tag, spacy.explain(tag)])

print(tabulate.tabulate(data, headers=['TAG', 'Description']))
```

```
TAG    Description
-----  -------------------------
ADJ    adjective
ADP    adposition
```

```
ADV    adverb
AUX    auxiliary
CCONJ  coordinating conjunction
DET    determiner
INTJ   interjection
NOUN   noun
NUM    numeral
PART   particle
PRON   pronoun
PROPN  proper noun
PUNCT  punctuation
SCONJ  subordinating conjunction
VERB   verb
X      other
```

It may also be useful to know the "confidence" of the tagger in making its decision. We can derive this from the output tagger's `model.predict` method. This returns (at least, in part) a ranked list of per-scores, the maximum value of which determines the final annotation.

```
# Helper function to get tagging scores

def get_tagging_scores(doc, n=3):
    # cf. https://stackoverflow.com/a/69228515
    scores = []
    tagger = nlp.get_pipe('tagger')
    labels = tagger.labels
    for token in doc:
        token_scores = tagger.model.predict([doc])[0][token.i]
        r = [*enumerate(token_scores)]
        r.sort(key=lambda x: x[1], reverse=True)
        scores.append([(labels[i], p) for i, p in r[:n]])
    return scores
```

```
# Get the top 3 tags by score for each token in the Doc

tagging_probs = get_tagging_scores(doc)

for token in doc:
    print(f'Token: {token.text}', end='\n\n')
    data = []
    for label, prob in tagging_probs[token.i]:
        data.append([label, prob])
```

```
    print(tabulate.tabulate(data, headers=['Label', 'Score']))
    break
```

```
Token: Haec

Label         Score
---------  --------
pronoun    10.7023
adjective   8.26614
noun        3.13534
```

References

NLTK Chapter 5 “Categorizing and tagging words” link
SLP Chapter 8 “Sequence labeling for parts of speech and named entities” link
spaCy Tagger and Morphologizer

8 Morphological Tagging

8.1 POS Tagging with LatinCy

Morphological tagging is the task of mapping a token in a text to various morphological tags as appropriate for the token's part of speech. A noun will have morphological tags for its gender, number, and case, while a verb—a finite verb, at least—will have tags for its person, number, tense, mood, and voice. The default LatinCy pipelines have a `morphologizer` component that assigns these tags.

```
# Imports & setup

import spacy
from pprint import pprint
nlp = spacy.load('la_core_web_lg')
text = "Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eiu
doc = nlp(text)
print(doc)
```

```
Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acrisi
```

Morphological tags are annotations of the `Token` objects and are stored in the `morph` attribute.

```
sample_token = doc[1]

print(f'Sample token: {sample_token.text}')
print(f'Sample morph: {sample_token.morph}')
```

```
Sample token: narrantur
Sample morph: Mood=Ind|Number=Plur|Person=3|Tense=Pres|VerbForm=Fin|Voice=Pass
```

Note that the morphological annotations are stored in a spaCy's `MorphAnalysis` object…

```
type(sample_token.morph)
```

```
spacy.tokens.morphanalysis.MorphAnalysis
```

LatinCy users may find the following `MorphAnalysis` methods of use.

To get all of the morphological tags for a token as a Python `dict`, you can use the `to_dict` method…

```
sample_morph_dict = sample_token.morph.to_dict()
pprint(sample_morph_dict)
```

```
{'Mood': 'Ind',
 'Number': 'Plur',
 'Person': '3',
 'Tense': 'Pres',
 'VerbForm': 'Fin',
 'Voice': 'Pass'}
```

You can also get the string representation of the morphological analysis with the `to_json` method:

```
sample_morph_dict = sample_token.morph.to_json()
pprint(sample_morph_dict)
```

```
'Mood=Ind|Number=Plur|Person=3|Tense=Pres|VerbForm=Fin|Voice=Pass'
```

Conversely, a `MorphAnalysis` object can be created from a Python `dict` using the `set_morph` method on a spaCy `Token` object.

```
text = "Festina lente."
doc_no_annotations = nlp.make_doc(text)
print(f'{doc_no_annotations[0].text} -> {doc_no_annotations[0].morph if doc_no_annotations[0]
```

```
Festina -> {No morph data}
```

```
festina_dict = {"Person": "2", "Number": "Singular", "Tense": "Present", "Mood": "Imperative"
doc_no_annotations[0].set_morph(festina_dict)
print(f'{doc_no_annotations[0].text} -> {doc_no_annotations[0].morph if doc_no_annotations[0]
print(type(doc_no_annotations[0].morph))
```

```
Festina -> Mood=Imperative|Number=Singular|Person=2|Tense=Present|Voice=Active
<class 'spacy.tokens.morphanalysis.MorphAnalysis'>
```

As noted above, the mophological tags are POS-specific. LatinCy uses a limited set of morphological keys based on the UD treebanks. Moreover, with respect to the tag values, one specific adjustment that has been made from the UD treebanks is that for verbs LatinCy uses the six "traditional" tense values, i.e. present, imperfect, future, perfect, pluperfect, and future perfect.

```
import tabulate

data = []

doc = nlp("Tum arcam ipsam in mare coniecit.")
tokens = [item for item in doc]

for token in tokens:
    data.append([token.text, token.morph.to_json()])

print(tabulate.tabulate(data, headers=['Text', "Morph"]))
```

```
Text      Morph
--------  ------------------------------------------------------------------
Tum
arcam     Case=Acc|Gender=Fem|Number=Sing
ipsam     Case=Acc|Gender=Fem|Number=Sing
in
mare      Case=Acc|Gender=Neut|Number=Sing
coniecit  Mood=Ind|Number=Sing|Person=3|Tense=Perf|VerbForm=Fin|Voice=Act
.
```

Note how in the sentence above the verb *coniecit* is tagged with the 'Tense=Perf' (for perfect), whereas in the UD treebanks this would have been annotated with 'Tense=Past' in coordination with 'Aspect=Perf'.

References

NLTK Chapter 5 "Categorizing and tagging words" link
spaCy Morphologizer

Part IV

spaCy Tasks

9 Sequence Matching

9.1 Sequence matching with LatinCy

We can use LatinCy annotations as the basis for matching spans of tokens using spaCy's `Matcher`. This includes basic `Token` attributes like `ORTH`, `TEXT`, `NORM`, and `LOWER` as well as those annotated by the LatinCy pipeline like `LEMMA`, `POS`, `TAG`, `MORPH`, `DEP`, and `ENT_TYPE`. There are also more general attributes like `IS_ALPHA`, `IS_ASCII`, `IS_DIGIT`, `IS_LOWER`, `IS_UPPER`, `IS_TITLE`, `IS_PUNCT`, `IS_SPACE`, and `LIKE_NUM`, among still others. The full list can be found here. Moreover, there are many operators, quantifiers, and other operations that can be using to create ever-increasingly complex patterns. The combinatorial patterns for using the `Matcher` are frankly enormous and so this should be considered only a most basic introduction for what is possible.

Here we run through a quick example based on finding variations of *res* and then by pattern-matching extensions of *res publica* using the *Praefatio* to Livy's *Ab urbe condita*.

```
# Imports & setup

import spacy
from pprint import pprint
from tabulate import tabulate

nlp = spacy.load('la_core_web_lg')

with open('livy_praefatio.txt') as f:
    text = f.read()

doc = nlp(text)
print(doc[:100])
```

```
Facturusne operae pretium sim si a primordio urbis res populi Romani perscripserim nec satis
```

The `Matcher` is initialized with the `Vocab` object from our loaded pipeline.

```
from spacy.matcher import Matcher

matcher = Matcher(nlp.vocab)
print(matcher)
```

```
<spacy.matcher.matcher.Matcher object at 0x157cc1e10>
```

We use the `Matcher` by adding patterns, quite sensibly using the `add` method. With the `add` method, we assign a `match_id` "name" for the pattern and the patterns themselves. The patters are lists of lists of dictionaries; those dictionaries are arranged sequentially by the tokens sequences we want to match, where the dictionary keys are the attributes and the dictionary values are the our specific terms to be matched for that attribute. So, in the example below, we are looking for any span of tokens in the provided `Doc` where the `text` attribute matches—and matches exactly—the string "res". The `Matcher` returns the `match_id` as well as the start and end indices for each matched span.

```
pattern = [{'TEXT': 'res'}]
matcher.add('res_tokens', [pattern])

matches = matcher(doc)

matches_data = []

for match_id, start, end in matches:
    string_id = nlp.vocab.strings[match_id]
    span = doc[start:end]
    matches_data.append((string_id, start, end, span.text))

print(tabulate(matches_data, headers=['Match ID', 'Start', 'End', 'Matched text']))
```

```
Match ID      Start    End  Matched text
----------  -------  -----  --------------
res_tokens        8      9  res
res_tokens      424    425  res
```

```
# Helper functions

def pattern2matches(pattern_name, pattern):
    matcher = Matcher(nlp.vocab)
    matcher.add(pattern_name, [pattern])
```

```
    matches = matcher(doc)
    matches_data = []
    for match_id, start, end in matches:
        string_id = nlp.vocab.strings[match_id]
        span = doc[start:end]
        matches_data.append((string_id, start, end, span.text))

    return matches_data

def tabulate_matches(pattern_name, pattern):
    matches_data = pattern2matches(pattern_name, pattern)
    print(tabulate(matches_data, headers=['Match ID', 'Start', 'End', 'Matched text']))
```

Note that the matches above on `TEXT` are case-sensitive. We can widen our search for *res* by using the `LOWER` attribute…

```
pattern = [{'LOWER': 'res'}]
tabulate_matches('res_uncased', pattern)
```

```
Match ID       Start    End  Matched text
-----------  -------  -----  --------------
res_uncased        8      9  res
res_uncased       93     94  Res
res_uncased      424    425  res
```

Extending this logic even further, we can widen the search again by matching not only the token "Res"/"res" but all tokens for which the LatinCy lemmatizers have assigned the lemma "res". This is done by using the `LEMMA` attribute.

```
pattern = [{'LEMMA': 'res'}]
tabulate_matches('res_lemma', pattern)
```

```
Match ID      Start    End  Matched text
----------  -------  -----  --------------
res_lemma         8      9  res
res_lemma        30     31  rem
res_lemma        39     40  rebus
res_lemma        57     58  rerum
res_lemma        93     94  Res
res_lemma       214    215  rerum
```

```
res_lemma       380    381  rerum
res_lemma       399    400  rei
res_lemma       424    425  res
res_lemma       463    464  rerum
res_lemma       507    508  rei
```

So far, all of our patterns have included only a single token. We can extend our search to include multiple sequential tokens by adding more dictionaries to the list of dictionaries. In the example below, we are looking for any span where the first token has the lemma "res" and is followed by any token with the `POS` of "NOUN".

```
pattern = [{'LEMMA': 'res'}, {'POS': 'NOUN'}]
tabulate_matches('res_lemma_noun', pattern)
```

```
Match ID          Start    End  Matched text
--------------  -------  -----  --------------
res_lemma_noun        8     10  res populi
```

By contrast, we can return a span where the first token has the lemma "res" and is *not* followed by any token with the `POS` of "NOUN".

```
pattern = [{'LEMMA': 'res'}, {'TAG': 'NOUN', "OP": "!"}]
tabulate_matches('res_lemma_noun_not', pattern)
```

```
Match ID              Start    End  Matched text
------------------  -------  -----  --------------
res_lemma_noun_not        8     10  res populi
res_lemma_noun_not       30     32  rem videam
res_lemma_noun_not       39     41  rebus certius
res_lemma_noun_not       57     59  rerum gestarum
res_lemma_noun_not       93     95  Res est
res_lemma_noun_not      214    216  rerum gestarum
res_lemma_noun_not      380    382  rerum salubre
res_lemma_noun_not      399    401  rei publicae
res_lemma_noun_not      424    426  res publica
res_lemma_noun_not      463    465  rerum minus
res_lemma_noun_not      507    509  rei absint
```

The `Matcher` allows for "fuzzy" matching based on Levenshtein distance (details in documentation, linked below)…

```
pattern = [{'LEMMA': 'res'}, {"TEXT": {"FUZZY": "public"}}]
tabulate_matches('res_public_fuzzy', pattern)
```

```
Match ID            Start    End  Matched text
----------------  -------  -----  --------------
res_public_fuzzy      399    401  rei publicae
res_public_fuzzy      424    426  res publica
```

Of course, you could search for *res publica* more directly with a two-lemma pattern…

```
pattern = [{'LEMMA': 'res'}, {'LEMMA': 'publicus'}]
tabulate_matches('res_publica_lemmas', pattern)
```

```
Match ID              Start    End  Matched text
------------------  -------  -----  --------------
res_publica_lemmas      399    401  rei publicae
res_publica_lemmas      424    426  res publica
```

The patterns are not limited to text search, that is they can be based entirely on annotation patterns. Here is a list of all NOUN-ADJ sequences in Livy's "Praefatio"…

```
pattern = [{"POS": "NOUN"}, {"POS": "ADJ"}]
tabulate_matches('noun_adjs', pattern)
```

```
Match ID      Start    End  Matched text
----------  -------  -----  ------------------
noun_adjs         9     11  populi Romani
noun_adjs        46     48  arte rudem
noun_adjs       127    129  origines proxima
noun_adjs       207    209  urbem poeticis
noun_adjs       237    239  urbium augustiora
noun_adjs       260    262  populo Romano
noun_adjs       276    278  gentes humanae
noun_adjs       380    382  rerum salubre
noun_adjs       399    401  rei publicae
noun_adjs       407    409  inceptu foedum
noun_adjs       424    426  res publica
noun_adjs       432    434  exemplis ditior
noun_adjs       539    541  successus prosperos
```

As well as a list of all alliterative patterns, though with some creative regexing…

```
matcher = Matcher(nlp.vocab)

for letter in "abcdefghijklmnopqrstuvwxyz":
    pattern = [{"LOWER": {"REGEX": rf'\b{letter}.+?\b'}, "OP": "{2,}"}]
    matcher.add('alliterative_pairs', [pattern])

matches = matcher(doc)

matches_data = []

for match_id, start, end in matches:
    string_id = nlp.vocab.strings[match_id]
    span = doc[start:end]
    matches_data.append((string_id, start, end, span.text))

print(tabulate(matches_data, headers=['Match ID', 'Start', 'End', 'Matched text']))
```

```
Match ID              Start    End  Matched text
------------------  -------  -----  --------------------------
alliterative_pairs        3      5  sim si
alliterative_pairs       13     15  satis scio
alliterative_pairs       17     19  si sciam
alliterative_pairs       23     25  quippe qui
alliterative_pairs       35     37  semper scriptores
alliterative_pairs       41     43  aliquid allaturos
alliterative_pairs       62     64  populi pro
alliterative_pairs      102    104  supra septingentesimum
alliterative_pairs      142    144  pridem praevalentis
alliterative_pairs      142    145  pridem praevalentis populi
alliterative_pairs      143    145  praevalentis populi
alliterative_pairs      155    157  praemium petam
alliterative_pairs      205    207  conditam condendamve
alliterative_pairs      279    281  aequo animo
alliterative_pairs      291    293  animaduersa aut
alliterative_pairs      293    295  existimata erunt
alliterative_pairs      347    349  magis magis
alliterative_pairs      365    367  nostra nec
alliterative_pairs      368    370  pati possumus
alliterative_pairs      368    371  pati possumus perventum
alliterative_pairs      369    371  possumus perventum
```

```
alliterative_pairs      389    391  in inlustri
alliterative_pairs      396    398  tibi tuae
alliterative_pairs      482    484  pereundi perdendi
alliterative_pairs      518    520  deorum dearum
```

References

spaCy Matcher

Open LatinCy Projects

Listed below are some open projects that I would like to see implemented in the LatinCy pipelines. If you are interested in contributing to the development of these projects or have recommendations for additional projects that could be added, please contact me through the LatinCy GitHub or HuggingFace repositories.

Model-based enclitic splitting

As discussed in the Word Tokenization chapter, enclitic splitting is currently limited to *que* (and variants) and is a rules-based process that is part of the LatinCy custom tokenizer. It would be preferable to make enclitic splitting a separate pipeline component and moreover one that is model-based. This component could be placed immediately after the tokenizer and use the `Retokenizer.split` method to reconstruct token sequences where valid enclitics are identified. This would have the added advantage of allowing enclitic splitting to be "turned off", so to speak, by removing the component from the pipeline.

References

Bird, Steven, Ewan Klein, and Edward Loper. 2015. *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. 2nd Edition. https://www.nltk.org/book/.

Burns, Patrick J. 2018. "Backoff Lemmatization as a Philological Method." In *Digital Humanities 2018, DH 2018, Book of Abstracts, El Colegio de México, UNAM, and RedHD, Mexico City, Mexico, June 26-29, 2018*.

———. 2020. "Ensemble Lemmatization with the Classical Language Toolkit." *Studi e Saggi Linguistici* 58 (1): 157–76. https://doi.org/10.4454/ssl.v58i1.273.

Clayman, Dee. 1981. "Sentence Length in Greek Hexameter Poetry." *Quantitative Linguistics* 11: 107–36. https://papers.ssrn.com/abstract=1627358.

Gamba, Federica, and Daniel Zeman. 2023. "Universalising Latin Universal Dependencies: A Harmonisation of Latin Treebanks in UD." In *Proceedings of the Sixth Workshop on Universal Dependencies (UDW, GURT/SyntaxFest 2023)*, edited by Loïc Grobol and Francis Tyers, 7–16. Washington, D.C.: Association for Computational Linguistics. https://aclanthology.org/2023.udw-1.2.

Janson, Tore. 1964. "The Problems of Measuring Sentence-Length in Classical Texts." *Studia Linguistica* 18 (1): 26–36. https://doi.org/10.1111/j.1467-9582.1964.tb00443.x.

Jurafsky, Daniel, and James H. Martin. 2020. "Speech and Language Processing (3rd Edition, Draft)." https://web.stanford.edu/~jurafsky/slp3/.

Wake, William C. 1957. "Sentence-Length Distributions of Greek Authors." *Journal of the Royal Statistical Society. Series A (General)* 120 (3): 331–46. https://www.jstor.org/stable/2343104.