File 1 - __init__.py:

```
1: (0)              from .blob import Blobber, Sentence, TextBlob, Word, WordList
2: (0)              __all__ = [
3: (4)                  "TextBlob",
4: (4)                  "Word",
5: (4)                  "Sentence",
6: (4)                  "Blobber",
7: (4)                  "WordList",
8: (0)              ]
```

File 2 - base.py:

```
1: (0)              """Abstract base classes for models (taggers, noun phrase extractors, etc.)
2: (0)              which define the interface for descendant classes.
3: (0)              .. versionchanged:: 0.7.0
4: (4)                  All base classes are defined in the same module, ``textblob.base``.
5: (0)              """
6: (0)              from abc import ABCMeta, abstractmethod
7: (0)              import nltk
8: (0)              class BaseTagger(metaclass=ABCMeta):
9: (4)                  """Abstract tagger class from which all taggers
10: (4)                 inherit from. All descendants must implement a
11: (4)                 ``tag()`` method.
12: (4)                 """
13: (4)                 @abstractmethod
14: (4)                 def tag(self, text, tokenize=True):
15: (8)                     """Return a list of tuples of the form (word, tag)
16: (8)                     for a given set of text or BaseBlob instance.
17: (8)                     """
18: (8)                     return
19: (0)             class BaseNPExtractor(metaclass=ABCMeta):
20: (4)                 """Abstract base class from which all NPExtractor classes inherit.
21: (4)                 Descendant classes must implement an ``extract(text)`` method
22: (4)                 that returns a list of noun phrases as strings.
23: (4)                 """
24: (4)                 @abstractmethod
25: (4)                 def extract(self, text):
26: (8)                     """Return a list of noun phrases (strings) for a body of text."""
27: (8)                     return
28: (0)             class BaseTokenizer(nltk.tokenize.api.TokenizerI, metaclass=ABCMeta):
29: (4)                 """Abstract base class from which all Tokenizer classes inherit.
30: (4)                 Descendant classes must implement a ``tokenize(text)`` method
31: (4)                 that returns a list of noun phrases as strings.
32: (4)                 """
33: (4)                 @abstractmethod
34: (4)                 def tokenize(self, text):
35: (8)                     """Return a list of tokens (strings) for a body of text.
36: (8)                     :rtype: list
37: (8)                     """
38: (8)                     return
39: (4)                 def itokenize(self, text, *args, **kwargs):
40: (8)                     """Return a generator that generates tokens "on-demand".
41: (8)                     .. versionadded:: 0.6.0
42: (8)                     :rtype: generator
43: (8)                     """
44: (8)                     return (t for t in self.tokenize(text, *args, **kwargs))
45: (0)             DISCRETE = "ds"
46: (0)             CONTINUOUS = "co"
47: (0)             class BaseSentimentAnalyzer(metaclass=ABCMeta):
48: (4)                 """Abstract base class from which all sentiment analyzers inherit.
49: (4)                 Should implement an ``analyze(text)`` method which returns either the
50: (4)                 results of analysis.
51: (4)                 """
52: (4)                 kind = DISCRETE
53: (4)                 def __init__(self):
54: (8)                     self._trained = False
```

```
55: (4)                 def train(self):
56: (8)                     self._trained = True
57: (4)                 @abstractmethod
58: (4)                 def analyze(self, text):
59: (8)                     """Return the result of of analysis. Typically returns either a
60: (8)                     tuple, float, or dictionary.
61: (8)                     """
62: (8)                     if not self._trained:
63: (12)                        self.train()
64: (8)                     return None
65: (0)             class BaseParser(metaclass=ABCMeta):
66: (4)                 """Abstract parser class from which all parsers inherit from. All
67: (4)                 descendants must implement a ``parse()`` method.
68: (4)                 """
69: (4)                 @abstractmethod
70: (4)                 def parse(self, text):
71: (8)                     """Parses the text."""
72: (8)                     return
```

File 3 - blob.py:

```
1: (0)              """Wrappers for various units of text, including the main
2: (0)              :class:`TextBlob <textblob.blob.TextBlob>`, :class:`Word
<textblob.blob.Word>`,
3: (0)              and :class:`WordList <textblob.blob.WordList>` classes.
4: (0)              Example usage: ::
5: (4)                  >>> from textblob import TextBlob
6: (4)                  >>> b = TextBlob("Simple is better than complex.")
7: (4)                  >>> b.tags
8: (4)                  [(u'Simple', u'NN'), (u'is', u'VBZ'), (u'better', u'JJR'), (u'than',
u'IN'), (u'complex', u'NN')]
9: (4)                  >>> b.noun_phrases
10: (4)                 WordList([u'simple'])
11: (4)                 >>> b.words
12: (4)                 WordList([u'Simple', u'is', u'better', u'than', u'complex'])
13: (4)                 >>> b.sentiment
14: (4)                 (0.06666666666666667, 0.41904761904761906)
15: (4)                 >>> b.words[0].synsets()[0]
16: (4)                 Synset('simple.n.01')
17: (0)             .. versionchanged:: 0.8.0
18: (4)                 These classes are now imported from ``textblob`` rather than
``text.blob``.
19: (0)             """  # noqa: E501
20: (0)             import json
21: (0)             import sys
22: (0)             from collections import defaultdict
23: (0)             import nltk
24: (0)             from textblob.base import (
25: (4)                 BaseNPExtractor,
26: (4)                 BaseParser,
27: (4)                 BaseSentimentAnalyzer,
28: (4)                 BaseTagger,
29: (4)                 BaseTokenizer,
30: (0)             )
31: (0)             from textblob.decorators import cached_property, requires_nltk_corpus
32: (0)             from textblob.en import suggest
33: (0)             from textblob.inflect import pluralize as _pluralize
34: (0)             from textblob.inflect import singularize as _singularize
35: (0)             from textblob.mixins import BlobComparableMixin, StringlikeMixin
36: (0)             from textblob.np_extractors import FastNPExtractor
37: (0)             from textblob.parsers import PatternParser
38: (0)             from textblob.sentiments import PatternAnalyzer
39: (0)             from textblob.taggers import NLTKTagger
40: (0)             from textblob.tokenizers import WordTokenizer, sent_tokenize, word_tokenize
41: (0)             from textblob.utils import PUNCTUATION_REGEX, lowerstrip
42: (0)             _wordnet = nltk.corpus.wordnet
43: (0)             basestring = (str, bytes)
```

```
 44: (0)             def _penn_to_wordnet(tag):
 45: (4)                 """Converts a Penn corpus tag into a Wordnet tag."""
 46: (4)                 if tag in ("NN", "NNS", "NNP", "NNPS"):
 47: (8)                     return _wordnet.NOUN
 48: (4)                 if tag in ("JJ", "JJR", "JJS"):
 49: (8)                     return _wordnet.ADJ
 50: (4)                 if tag in ("VB", "VBD", "VBG", "VBN", "VBP", "VBZ"):
 51: (8)                     return _wordnet.VERB
 52: (4)                 if tag in ("RB", "RBR", "RBS"):
 53: (8)                     return _wordnet.ADV
 54: (4)                 return None
 55: (0)             class Word(str):
 56: (4)                 """A simple word representation. Includes methods for inflection,
 57: (4)                 translation, and WordNet integration.
 58: (4)                 """
 59: (4)                 def __new__(cls, string, pos_tag=None):
 60: (8)                     """Return a new instance of the class. It is necessary to override
 61: (8)                     this method in order to handle the extra pos_tag argument in the
 62: (8)                     constructor.
 63: (8)                     """
 64: (8)                     return super().__new__(cls, string)
 65: (4)                 def __init__(self, string, pos_tag=None):
 66: (8)                     self.string = string
 67: (8)                     self.pos_tag = pos_tag
 68: (4)                 def __repr__(self):
 69: (8)                     return repr(self.string)
 70: (4)                 def __str__(self):
 71: (8)                     return self.string
 72: (4)                 def singularize(self):
 73: (8)                     """Return the singular version of the word as a string."""
 74: (8)                     return Word(_singularize(self.string))
 75: (4)                 def pluralize(self):
 76: (8)                     """Return the plural version of the word as a string."""
 77: (8)                     return Word(_pluralize(self.string))
 78: (4)                 def spellcheck(self):
 79: (8)                     """Return a list of (word, confidence) tuples of spelling corrections.
 80: (8)                     Based on: Peter Norvig, "How to Write a Spelling Corrector"
 81: (8)                     (http://norvig.com/spell-correct.html) as implemented in the pattern
 82: (8)                     library.
 83: (8)                     .. versionadded:: 0.6.0
 84: (8)                     """
 85: (8)                     return suggest(self.string)
 86: (4)                 def correct(self):
 87: (8)                     """Correct the spelling of the word. Returns the word with the highest
 88: (8)                     confidence using the spelling corrector.
 89: (8)                     .. versionadded:: 0.6.0
 90: (8)                     """
 91: (8)                     return Word(self.spellcheck()[0][0])
 92: (4)                 @cached_property
 93: (4)                 @requires_nltk_corpus
 94: (4)                 def lemma(self):
 95: (8)                     """Return the lemma of this word using Wordnet's morphy function."""
 96: (8)                     return self.lemmatize(pos=self.pos_tag)
 97: (4)                 @requires_nltk_corpus
 98: (4)                 def lemmatize(self, pos=None):
 99: (8)                     """Return the lemma for a word using WordNet's morphy function.
100: (8)                     :param pos: Part of speech to filter upon. If `None`, defaults to
101: (12)                        ``_wordnet.NOUN``.
102: (8)                     .. versionadded:: 0.8.1
103: (8)                     """
104: (8)                     if pos is None:
105: (12)                        tag = _wordnet.NOUN
106: (8)                     elif pos in _wordnet._FILEMAP.keys():
107: (12)                        tag = pos
108: (8)                     else:
109: (12)                        tag = _penn_to_wordnet(pos)
110: (8)                     lemmatizer = nltk.stem.WordNetLemmatizer()
111: (8)                     return lemmatizer.lemmatize(self.string, tag)
112: (4)                 PorterStemmer = nltk.stem.porter.PorterStemmer()
```

```
113: (4)                LancasterStemmer = nltk.stem.lancaster.LancasterStemmer()
114: (4)                SnowballStemmer = nltk.stem.snowball.SnowballStemmer("english")
115: (4)                def stem(self, stemmer=PorterStemmer):
116: (8)                    """Stem a word using various NLTK stemmers. (Default: Porter Stemmer)
117: (8)                    .. versionadded:: 0.12.0
118: (8)                    """
119: (8)                    return stemmer.stem(self.string)
120: (4)                @cached_property
121: (4)                def synsets(self):
122: (8)                    """The list of Synset objects for this Word.
123: (8)                    :rtype: list of Synsets
124: (8)                    .. versionadded:: 0.7.0
125: (8)                    """
126: (8)                    return self.get_synsets(pos=None)
127: (4)                @cached_property
128: (4)                def definitions(self):
129: (8)                    """The list of definitions for this word. Each definition corresponds
130: (8)                    to a synset.
131: (8)                    .. versionadded:: 0.7.0
132: (8)                    """
133: (8)                    return self.define(pos=None)
134: (4)                def get_synsets(self, pos=None):
135: (8)                    """Return a list of Synset objects for this word.
136: (8)                    :param pos: A part-of-speech tag to filter upon. If ``None``, all
137: (12)                       synsets for all parts of speech will be loaded.
138: (8)                    :rtype: list of Synsets
139: (8)                    .. versionadded:: 0.7.0
140: (8)                    """
141: (8)                    return _wordnet.synsets(self.string, pos)
142: (4)                def define(self, pos=None):
143: (8)                    """Return a list of definitions for this word. Each definition
144: (8)                    corresponds to a synset for this word.
145: (8)                    :param pos: A part-of-speech tag to filter upon. If ``None``,
definitions
146: (12)                       for all parts of speech will be loaded.
147: (8)                    :rtype: List of strings
148: (8)                    .. versionadded:: 0.7.0
149: (8)                    """
150: (8)                    return [syn.definition() for syn in self.get_synsets(pos=pos)]
151: (0)            class WordList(list):
152: (4)                """A list-like collection of words."""
153: (4)                def __init__(self, collection):
154: (8)                    """Initialize a WordList. Takes a collection of strings as
155: (8)                    its only argument.
156: (8)                    """
157: (8)                    super().__init__([Word(w) for w in collection])
158: (4)                def __str__(self):
159: (8)                    """Returns a string representation for printing."""
160: (8)                    return super().__repr__()
161: (4)                def __repr__(self):
162: (8)                    """Returns a string representation for debugging."""
163: (8)                    class_name = self.__class__.__name__
164: (8)                    return f"{class_name}({super().__repr__()})"
165: (4)                def __getitem__(self, key):
166: (8)                    """Returns a string at the given index."""
167: (8)                    item = super().__getitem__(key)
168: (8)                    if isinstance(key, slice):
169: (12)                       return self.__class__(item)
170: (8)                    else:
171: (12)                       return item
172: (4)                def __getslice__(self, i, j):
173: (8)                    return self.__class__(super().__getslice__(i, j))
174: (4)                def __setitem__(self, index, obj):
175: (8)                    """Places object at given index, replacing existing item. If the
object
176: (8)                    is a string, inserts a :class:`Word <Word>` object.
177: (8)                    """
178: (8)                    if isinstance(obj, basestring):
179: (12)                       super().__setitem__(index, Word(obj))
```

```
180: (8)                    else:
181: (12)                       super().__setitem__(index, obj)
182: (4)                def count(self, strg, case_sensitive=False, *args, **kwargs):
183: (8)                    """Get the count of a word or phrase `s` within this WordList.
184: (8)                    :param strg: The string to count.
185: (8)                    :param case_sensitive: A boolean, whether or not the search is case-
sensitive.
186: (8)                    """
187: (8)                    if not case_sensitive:
188: (12)                       return [word.lower() for word in self].count(strg.lower(), *args,
**kwargs)
189: (8)                    return super().count(strg, *args, **kwargs)
190: (4)                def append(self, obj):
191: (8)                    """Append an object to end. If the object is a string, appends a
192: (8)                    :class:`Word <Word>` object.
193: (8)                    """
194: (8)                    if isinstance(obj, basestring):
195: (12)                       super().append(Word(obj))
196: (8)                    else:
197: (12)                       super().append(obj)
198: (4)                def extend(self, iterable):
199: (8)                    """Extend WordList by appending elements from ``iterable``. If an
element
200: (8)                    is a string, appends a :class:`Word <Word>` object.
201: (8)                    """
202: (8)                    for e in iterable:
203: (12)                       self.append(e)
204: (4)                def upper(self):
205: (8)                    """Return a new WordList with each word upper-cased."""
206: (8)                    return self.__class__([word.upper() for word in self])
207: (4)                def lower(self):
208: (8)                    """Return a new WordList with each word lower-cased."""
209: (8)                    return self.__class__([word.lower() for word in self])
210: (4)                def singularize(self):
211: (8)                    """Return the single version of each word in this WordList."""
212: (8)                    return self.__class__([word.singularize() for word in self])
213: (4)                def pluralize(self):
214: (8)                    """Return the plural version of each word in this WordList."""
215: (8)                    return self.__class__([word.pluralize() for word in self])
216: (4)                def lemmatize(self):
217: (8)                    """Return the lemma of each word in this WordList."""
218: (8)                    return self.__class__([word.lemmatize() for word in self])
219: (4)                def stem(self, *args, **kwargs):
220: (8)                    """Return the stem for each word in this WordList."""
221: (8)                    return self.__class__([word.stem(*args, **kwargs) for word in self])
222: (0)            def _validated_param(obj, name, base_class, default, base_class_name=None):
223: (4)                """Validates a parameter passed to __init__. Makes sure that obj is
224: (4)                the correct class. Return obj if it's not None or falls back to default
225: (4)                :param obj: The object passed in.
226: (4)                :param name: The name of the parameter.
227: (4)                :param base_class: The class that obj must inherit from.
228: (4)                :param default: The default object to fall back upon if obj is None.
229: (4)                """
230: (4)                base_class_name = base_class_name if base_class_name else
base_class.__name__
231: (4)                if obj is not None and not isinstance(obj, base_class):
232: (8)                    raise ValueError(f"{name} must be an instance of {base_class_name}")
233: (4)                return obj or default
234: (0)            def _initialize_models(
235: (4)                obj, tokenizer, pos_tagger, np_extractor, analyzer, parser, classifier
236: (0)            ):
237: (4)                """Common initialization between BaseBlob and Blobber classes."""
238: (4)                obj.tokenizer = _validated_param(
239: (8)                    tokenizer,
240: (8)                    "tokenizer",
241: (8)                    base_class=(BaseTokenizer, nltk.tokenize.api.TokenizerI),
242: (8)                    default=BaseBlob.tokenizer,
243: (8)                    base_class_name="BaseTokenizer",
244: (4)                )
```

```
245: (4)                obj.np_extractor = _validated_param(
246: (8)                    np_extractor,
247: (8)                    "np_extractor",
248: (8)                    base_class=BaseNPExtractor,
249: (8)                    default=BaseBlob.np_extractor,
250: (4)                )
251: (4)                obj.pos_tagger = _validated_param(
252: (8)                    pos_tagger, "pos_tagger", BaseTagger, BaseBlob.pos_tagger
253: (4)                )
254: (4)                obj.analyzer = _validated_param(
255: (8)                    analyzer, "analyzer", BaseSentimentAnalyzer, BaseBlob.analyzer
256: (4)                )
257: (4)                obj.parser = _validated_param(parser, "parser", BaseParser,
BaseBlob.parser)
258: (4)                obj.classifier = classifier
259: (0)            class BaseBlob(StringlikeMixin, BlobComparableMixin):
260: (4)                """An abstract base class that all textblob classes will inherit from.
261: (4)                Includes words, POS tag, NP, and word count properties. Also includes
262: (4)                basic dunder and string methods for making objects like Python strings.
263: (4)                :param text: A string.
264: (4)                :param tokenizer: (optional) A tokenizer instance. If ``None``,
265: (8)                    defaults to :class:`WordTokenizer()
<textblob.tokenizers.WordTokenizer>`.
266: (4)                :param np_extractor: (optional) An NPExtractor instance. If ``None``,
267: (8)                    defaults to :class:`FastNPExtractor()
<textblob.en.np_extractors.FastNPExtractor>`.
268: (4)                :param pos_tagger: (optional) A Tagger instance. If ``None``,
269: (8)                    defaults to :class:`NLTKTagger <textblob.en.taggers.NLTKTagger>`.
270: (4)                :param analyzer: (optional) A sentiment analyzer. If ``None``,
271: (8)                    defaults to :class:`PatternAnalyzer
<textblob.en.sentiments.PatternAnalyzer>`.
272: (4)                :param parser: A parser. If ``None``, defaults to
273: (8)                    :class:`PatternParser <textblob.en.parsers.PatternParser>`.
274: (4)                :param classifier: A classifier.
275: (4)                .. versionchanged:: 0.6.0
276: (8)                    ``clean_html`` parameter deprecated, as it was in NLTK.
277: (4)                """  # noqa: E501
278: (4)                np_extractor = FastNPExtractor()
279: (4)                pos_tagger = NLTKTagger()
280: (4)                tokenizer = WordTokenizer()
281: (4)                analyzer = PatternAnalyzer()
282: (4)                parser = PatternParser()
283: (4)                def __init__(
284: (8)                    self,
285: (8)                    text,
286: (8)                    tokenizer=None,
287: (8)                    pos_tagger=None,
288: (8)                    np_extractor=None,
289: (8)                    analyzer=None,
290: (8)                    parser=None,
291: (8)                    classifier=None,
292: (8)                    clean_html=False,
293: (4)                ):
294: (8)                    if not isinstance(text, basestring):
295: (12)                       raise TypeError(
296: (16)                           "The `text` argument passed to `__init__(text)` "
297: (16)                           f"must be a string, not {type(text)}"
298: (12)                       )
299: (8)                    if clean_html:
300: (12)                       raise NotImplementedError(
301: (16)                           "clean_html has been deprecated. "
302: (16)                           "To remove HTML markup, use BeautifulSoup's "
303: (16)                           "get_text() function"
304: (12)                       )
305: (8)                    self.raw = self.string = text
306: (8)                    self.stripped = lowerstrip(self.raw, all=True)
307: (8)                    _initialize_models(
308: (12)                       self, tokenizer, pos_tagger, np_extractor, analyzer, parser,
classifier
```

```
309: (8)                    )
310: (4)                @cached_property
311: (4)                def words(self):
312: (8)                    """Return a list of word tokens. This excludes punctuation characters.
313: (8)                    If you want to include punctuation characters, access the ``tokens``
314: (8)                    property.
315: (8)                    :returns: A :class:`WordList <WordList>` of word tokens.
316: (8)                    """
317: (8)                    return WordList(word_tokenize(self.raw, include_punc=False))
318: (4)                @cached_property
319: (4)                def tokens(self):
320: (8)                    """Return a list of tokens, using this blob's tokenizer object
321: (8)                    (defaults to :class:`WordTokenizer
<textblob.tokenizers.WordTokenizer>`).
322: (8)                    """
323: (8)                    return WordList(self.tokenizer.tokenize(self.raw))
324: (4)                def tokenize(self, tokenizer=None):
325: (8)                    """Return a list of tokens, using ``tokenizer``.
326: (8)                    :param tokenizer: (optional) A tokenizer object. If None, defaults to
327: (12)                       this blob's default tokenizer.
328: (8)                    """
329: (8)                    t = tokenizer if tokenizer is not None else self.tokenizer
330: (8)                    return WordList(t.tokenize(self.raw))
331: (4)                def parse(self, parser=None):
332: (8)                    """Parse the text.
333: (8)                    :param parser: (optional) A parser instance. If ``None``, defaults to
334: (12)                       this blob's default parser.
335: (8)                    .. versionadded:: 0.6.0
336: (8)                    """
337: (8)                    p = parser if parser is not None else self.parser
338: (8)                    return p.parse(self.raw)
339: (4)                def classify(self):
340: (8)                    """Classify the blob using the blob's ``classifier``."""
341: (8)                    if self.classifier is None:
342: (12)                       raise NameError("This blob has no classifier. Train one first!")
343: (8)                    return self.classifier.classify(self.raw)
344: (4)                @cached_property
345: (4)                def sentiment(self):
346: (8)                    """Return a tuple of form (polarity, subjectivity ) where polarity
347: (8)                    is a float within the range [-1.0, 1.0] and subjectivity is a float
348: (8)                    within the range [0.0, 1.0] where 0.0 is very objective and 1.0 is
349: (8)                    very subjective.
350: (8)                    :rtype: namedtuple of the form ``Sentiment(polarity, subjectivity)``
351: (8)                    """
352: (8)                    return self.analyzer.analyze(self.raw)
353: (4)                @cached_property
354: (4)                def sentiment_assessments(self):
355: (8)                    """Return a tuple of form (polarity, subjectivity, assessments ) where
356: (8)                    polarity is a float within the range [-1.0, 1.0], subjectivity is a
357: (8)                    float within the range [0.0, 1.0] where 0.0 is very objective and 1.0
358: (8)                    is very subjective, and assessments is a list of polarity and
359: (8)                    subjectivity scores for the assessed tokens.
360: (8)                    :rtype: namedtuple of the form ``Sentiment(polarity, subjectivity,
361: (8)                    assessments)``
362: (8)                    """
363: (8)                    return self.analyzer.analyze(self.raw, keep_assessments=True)
364: (4)                @cached_property
365: (4)                def polarity(self):
366: (8)                    """Return the polarity score as a float within the range [-1.0, 1.0]
367: (8)                    :rtype: float
368: (8)                    """
369: (8)                    return PatternAnalyzer().analyze(self.raw)[0]
370: (4)                @cached_property
371: (4)                def subjectivity(self):
372: (8)                    """Return the subjectivity score as a float within the range [0.0,
1.0]
373: (8)                    where 0.0 is very objective and 1.0 is very subjective.
374: (8)                    :rtype: float
375: (8)                    """
```

```
376: (8)                 return PatternAnalyzer().analyze(self.raw)[1]
377: (4)             @cached_property
378: (4)             def noun_phrases(self):
379: (8)                 """Returns a list of noun phrases for this blob."""
380: (8)                 return WordList(
381: (12)                    [
382: (16)                        phrase.strip().lower()
383: (16)                        for phrase in self.np_extractor.extract(self.raw)
384: (16)                        if len(phrase) > 1
385: (12)                    ]
386: (8)                 )
387: (4)             @cached_property
388: (4)             def pos_tags(self):
389: (8)                 """Returns an list of tuples of the form (word, POS tag).
390: (8)                 Example:
391: (8)                 ::
392: (12)                    [('At', 'IN'), ('eight', 'CD'), ("o'clock", 'JJ'), ('on', 'IN'),
393: (20)                            ('Thursday', 'NNP'), ('morning', 'NN')]
394: (8)                 :rtype: list of tuples
395: (8)                 """
396: (8)                 if isinstance(self, TextBlob):
397: (12)                    return [
398: (16)                        val
399: (16)                        for sublist in [s.pos_tags for s in self.sentences]
400: (16)                        for val in sublist
401: (12)                    ]
402: (8)                 else:
403: (12)                    return [
404: (16)                        (Word(str(word), pos_tag=t), str(t))
405: (16)                        for word, t in self.pos_tagger.tag(self)
406: (16)                        if not PUNCTUATION_REGEX.match(str(t))
407: (12)                    ]
408: (4)             tags = pos_tags
409: (4)             @cached_property
410: (4)             def word_counts(self):
411: (8)                 """Dictionary of word frequencies in this text."""
412: (8)                 counts = defaultdict(int)
413: (8)                 stripped_words = [lowerstrip(word) for word in self.words]
414: (8)                 for word in stripped_words:
415: (12)                    counts[word] += 1
416: (8)                 return counts
417: (4)             @cached_property
418: (4)             def np_counts(self):
419: (8)                 """Dictionary of noun phrase frequencies in this text."""
420: (8)                 counts = defaultdict(int)
421: (8)                 for phrase in self.noun_phrases:
422: (12)                    counts[phrase] += 1
423: (8)                 return counts
424: (4)             def ngrams(self, n=3):
425: (8)                 """Return a list of n-grams (tuples of n successive words) for this
426: (8)                 blob.
427: (8)                 :rtype: List of :class:`WordLists <WordList>`
428: (8)                 """
429: (8)                 if n <= 0:
430: (12)                    return []
431: (8)                 grams = [
432: (12)                    WordList(self.words[i : i + n]) for i in range(len(self.words) - n
+ 1)
433: (8)                 ]
434: (8)                 return grams
435: (4)             def correct(self):
436: (8)                 """Attempt to correct the spelling of a blob.
437: (8)                 .. versionadded:: 0.6.0
438: (8)                 :rtype: :class:`BaseBlob <BaseBlob>`
439: (8)                 """
440: (8)                 tokens = nltk.tokenize.regexp_tokenize(self.raw, r"\w+|[^\w\s]|\s")
441: (8)                 corrected = (Word(w).correct() for w in tokens)
442: (8)                 ret = "".join(corrected)
443: (8)                 return self.__class__(ret)
```

```
444: (4)                def _cmpkey(self):
445: (8)                    """Key used by ComparableMixin to implement all rich comparison
446: (8)                    operators.
447: (8)                    """
448: (8)                    return self.raw
449: (4)                def _strkey(self):
450: (8)                    """Key used by StringlikeMixin to implement string methods."""
451: (8)                    return self.raw
452: (4)                def __hash__(self):
453: (8)                    return hash(self._cmpkey())
454: (4)                def __add__(self, other):
455: (8)                    """Concatenates two text objects the same way Python strings are
456: (8)                    concatenated.
457: (8)                    Arguments:
458: (8)                    - `other`: a string or a text object
459: (8)                    """
460: (8)                    if isinstance(other, basestring):
461: (12)                       return self.__class__(self.raw + other)
462: (8)                    elif isinstance(other, BaseBlob):
463: (12)                       return self.__class__(self.raw + other.raw)
464: (8)                    else:
465: (12)                       raise TypeError(
466: (16)                           f"Operands must be either strings or {self.__class__.__name__}
objects"
467: (12)                       )
468: (4)                def split(self, sep=None, maxsplit=sys.maxsize):
469: (8)                    """Behaves like the built-in str.split() except returns a
470: (8)                    WordList.
471: (8)                    :rtype: :class:`WordList <WordList>`
472: (8)                    """
473: (8)                    return WordList(self._strkey().split(sep, maxsplit))
474: (0)            class TextBlob(BaseBlob):
475: (4)                """A general text block, meant for larger bodies of text (esp. those
476: (4)                containing sentences). Inherits from :class:`BaseBlob <BaseBlob>`.
477: (4)                :param str text: A string.
478: (4)                :param tokenizer: (optional) A tokenizer instance. If ``None``, defaults
to
479: (8)                    :class:`WordTokenizer() <textblob.tokenizers.WordTokenizer>`.
480: (4)                :param np_extractor: (optional) An NPExtractor instance. If ``None``,
481: (8)                    defaults to :class:`FastNPExtractor()
<textblob.en.np_extractors.FastNPExtractor>`.
482: (4)                :param pos_tagger: (optional) A Tagger instance. If ``None``, defaults to
483: (8)                    :class:`NLTKTagger <textblob.en.taggers.NLTKTagger>`.
484: (4)                :param analyzer: (optional) A sentiment analyzer. If ``None``, defaults to
485: (8)                    :class:`PatternAnalyzer <textblob.en.sentiments.PatternAnalyzer>`.
486: (4)                :param classifier: (optional) A classifier.
487: (4)                """  # noqa: E501
488: (4)                @cached_property
489: (4)                def sentences(self):
490: (8)                    """Return list of :class:`Sentence <Sentence>` objects."""
491: (8)                    return self._create_sentence_objects()
492: (4)                @cached_property
493: (4)                def words(self):
494: (8)                    """Return a list of word tokens. This excludes punctuation characters.
495: (8)                    If you want to include punctuation characters, access the ``tokens``
496: (8)                    property.
497: (8)                    :returns: A :class:`WordList <WordList>` of word tokens.
498: (8)                    """
499: (8)                    return WordList(word_tokenize(self.raw, include_punc=False))
500: (4)                @property
501: (4)                def raw_sentences(self):
502: (8)                    """List of strings, the raw sentences in the blob."""
503: (8)                    return [sentence.raw for sentence in self.sentences]
504: (4)                @property
505: (4)                def serialized(self):
506: (8)                    """Returns a list of each sentence's dict representation."""
507: (8)                    return [sentence.dict for sentence in self.sentences]
508: (4)                def to_json(self, *args, **kwargs):
509: (8)                    """Return a json representation (str) of this blob.
```

```
510: (8)                    Takes the same arguments as json.dumps.
511: (8)                    .. versionadded:: 0.5.1
512: (8)                    """
513: (8)                    return json.dumps(self.serialized, *args, **kwargs)
514: (4)                @property
515: (4)                def json(self):
516: (8)                    """The json representation of this blob.
517: (8)                    .. versionchanged:: 0.5.1
518: (12)                       Made ``json`` a property instead of a method to restore backwards
519: (12)                       compatibility that was broken after version 0.4.0.
520: (8)                    """
521: (8)                    return self.to_json()
522: (4)                def _create_sentence_objects(self):
523: (8)                    """Returns a list of Sentence objects from the raw text."""
524: (8)                    sentence_objects = []
525: (8)                    sentences = sent_tokenize(self.raw)
526: (8)                    char_index = 0  # Keeps track of character index within the blob
527: (8)                    for sent in sentences:
528: (12)                       start_index = self.raw.index(sent, char_index)
529: (12)                       char_index += len(sent)
530: (12)                       end_index = start_index + len(sent)
531: (12)                       s = Sentence(
532: (16)                           sent,
533: (16)                           start_index=start_index,
534: (16)                           end_index=end_index,
535: (16)                           tokenizer=self.tokenizer,
536: (16)                           np_extractor=self.np_extractor,
537: (16)                           pos_tagger=self.pos_tagger,
538: (16)                           analyzer=self.analyzer,
539: (16)                           parser=self.parser,
540: (16)                           classifier=self.classifier,
541: (12)                       )
542: (12)                       sentence_objects.append(s)
543: (8)                    return sentence_objects
544: (0)            class Sentence(BaseBlob):
545: (4)                """A sentence within a TextBlob. Inherits from :class:`BaseBlob
    <BaseBlob>`.
546: (4)                :param sentence: A string, the raw sentence.
547: (4)                :param start_index: An int, the index where this sentence begins
548: (24)                                    in a TextBlob. If not given, defaults to 0.
549: (4)                :param end_index: An int, the index where this sentence ends in
550: (24)                                   a TextBlob. If not given, defaults to the
551: (24)                                   length of the sentence - 1.
552: (4)                """
553: (4)                def __init__(self, sentence, start_index=0, end_index=None, *args,
    **kwargs):
554: (8)                    super().__init__(sentence, *args, **kwargs)
555: (8)                    self.start = self.start_index = start_index
556: (8)                    self.end = self.end_index = end_index or len(sentence) - 1
557: (4)                @property
558: (4)                def dict(self):
559: (8)                    """The dict representation of this sentence."""
560: (8)                    return {
561: (12)                       "raw": self.raw,
562: (12)                       "start_index": self.start_index,
563: (12)                       "end_index": self.end_index,
564: (12)                       "stripped": self.stripped,
565: (12)                       "noun_phrases": self.noun_phrases,
566: (12)                       "polarity": self.polarity,
567: (12)                       "subjectivity": self.subjectivity,
568: (8)                    }
569: (0)            class Blobber:
570: (4)                """A factory for TextBlobs that all share the same tagger,
571: (4)                tokenizer, parser, classifier, and np_extractor.
572: (4)                Usage:
573: (8)                    >>> from textblob import Blobber
574: (8)                    >>> from textblob.taggers import NLTKTagger
575: (8)                    >>> from textblob.tokenizers import SentenceTokenizer
576: (8)                    >>> tb = Blobber(pos_tagger=NLTKTagger(),
```

```
tokenizer=SentenceTokenizer())
577: (8)                    >>> blob1 = tb("This is one blob.")
578: (8)                    >>> blob2 = tb("This blob has the same tagger and tokenizer.")
579: (8)                    >>> blob1.pos_tagger is blob2.pos_tagger
580: (8)                    True
581: (4)                :param tokenizer: (optional) A tokenizer instance. If ``None``,
582: (8)                    defaults to :class:`WordTokenizer()
<textblob.tokenizers.WordTokenizer>`.
583: (4)                :param np_extractor: (optional) An NPExtractor instance. If ``None``,
584: (8)                    defaults to :class:`FastNPExtractor()
<textblob.en.np_extractors.FastNPExtractor>`.
585: (4)                :param pos_tagger: (optional) A Tagger instance. If ``None``,
586: (8)                    defaults to :class:`NLTKTagger <textblob.en.taggers.NLTKTagger>`.
587: (4)                :param analyzer: (optional) A sentiment analyzer. If ``None``,
588: (8)                    defaults to :class:`PatternAnalyzer
<textblob.en.sentiments.PatternAnalyzer>`.
589: (4)                :param parser: A parser. If ``None``, defaults to
590: (8)                    :class:`PatternParser <textblob.en.parsers.PatternParser>`.
591: (4)                :param classifier: A classifier.
592: (4)                .. versionadded:: 0.4.0
593: (4)                """  # noqa: E501
594: (4)                np_extractor = FastNPExtractor()
595: (4)                pos_tagger = NLTKTagger()
596: (4)                tokenizer = WordTokenizer()
597: (4)                analyzer = PatternAnalyzer()
598: (4)                parser = PatternParser()
599: (4)                def __init__(
600: (8)                    self,
601: (8)                    tokenizer=None,
602: (8)                    pos_tagger=None,
603: (8)                    np_extractor=None,
604: (8)                    analyzer=None,
605: (8)                    parser=None,
606: (8)                    classifier=None,
607: (4)                ):
608: (8)                    _initialize_models(
609: (12)                       self, tokenizer, pos_tagger, np_extractor, analyzer, parser,
classifier
610: (8)                    )
611: (4)                def __call__(self, text):
612: (8)                    """Return a new TextBlob object with this Blobber's ``np_extractor``,
613: (8)                    ``pos_tagger``, ``tokenizer``, ``analyzer``, and ``classifier``.
614: (8)                    :returns: A new :class:`TextBlob <TextBlob>`.
615: (8)                    """
616: (8)                    return TextBlob(
617: (12)                       text,
618: (12)                       tokenizer=self.tokenizer,
619: (12)                       pos_tagger=self.pos_tagger,
620: (12)                       np_extractor=self.np_extractor,
621: (12)                       analyzer=self.analyzer,
622: (12)                       parser=self.parser,
623: (12)                       classifier=self.classifier,
624: (8)                    )
625: (4)                def __repr__(self):
626: (8)                    classifier_name = (
627: (12)                       self.classifier.__class__.__name__ + "()" if self.classifier else
"None"
628: (8)                    )
629: (8)                    return (
630: (12)                       "Blobber(tokenizer={}(), pos_tagger={}(), "
631: (12)                       "np_extractor={}(), analyzer={}(), parser={}(), classifier={})"
632: (8)                    ).format(
633: (12)                       self.tokenizer.__class__.__name__,
634: (12)                       self.pos_tagger.__class__.__name__,
635: (12)                       self.np_extractor.__class__.__name__,
636: (12)                       self.analyzer.__class__.__name__,
637: (12)                       self.parser.__class__.__name__,
638: (12)                       classifier_name,
639: (8)                    )
```

```
640: (4)                __str__ = __repr__

----------------------------------------

File 4 - _text.py:

1: (0)              """This file is adapted from the pattern library.
2: (0)              URL: http://www.clips.ua.ac.be/pages/pattern-web
3: (0)              Licence: BSD
4: (0)              """
5: (0)              import codecs
6: (0)              import os
7: (0)              import re
8: (0)              import string
9: (0)              import types
10: (0)             from itertools import chain
11: (0)             from xml.etree import ElementTree
12: (0)             basestring = (str, bytes)
13: (0)             try:
14: (4)                 MODULE = os.path.dirname(os.path.abspath(__file__))
15: (0)             except:
16: (4)                 MODULE = ""
17: (0)             SLASH, WORD, POS, CHUNK, PNP, REL, ANCHOR, LEMMA = (
18: (4)                 "&slash;",
19: (4)                 "word",
20: (4)                 "part-of-speech",
21: (4)                 "chunk",
22: (4)                 "preposition",
23: (4)                 "relation",
24: (4)                 "anchor",
25: (4)                 "lemma",
26: (0)             )
27: (0)             def decode_string(v, encoding="utf-8"):
28: (4)                 """Returns the given value as a Unicode string (if possible)."""
29: (4)                 if isinstance(encoding, basestring):
30: (8)                     encoding = ((encoding,),) + (("windows-1252",), ("utf-8", "ignore"))
31: (4)                 if isinstance(v, bytes):
32: (8)                     for e in encoding:
33: (12)                        try:
34: (16)                            return v.decode(*e)
35: (12)                        except:
36: (16)                            pass
37: (8)                     return v
38: (4)                 return str(v)
39: (0)             def encode_string(v, encoding="utf-8"):
40: (4)                 """Returns the given value as a Python byte string (if possible)."""
41: (4)                 if isinstance(encoding, basestring):
42: (8)                     encoding = ((encoding,),) + (("windows-1252",), ("utf-8", "ignore"))
43: (4)                 if isinstance(v, str):
44: (8)                     for e in encoding:
45: (12)                        try:
46: (16)                            return v.encode(*e)
47: (12)                        except:
48: (16)                            pass
49: (8)                     return v
50: (4)                 return str(v)
51: (0)             decode_utf8 = decode_string
52: (0)             encode_utf8 = encode_string
53: (0)             def isnumeric(strg):
54: (4)                 try:
55: (8)                     float(strg)
56: (4)                 except ValueError:
57: (8)                     return False
58: (4)                 return True
59: (0)             class lazydict(dict):
60: (4)                 def load(self):
61: (8)                     pass
62: (4)                 def _lazy(self, method, *args):
63: (8)                     """If the dictionary is empty, calls lazydict.load().
```

```
64: (8)                     Replaces lazydict.method() with dict.method() and calls it.
65: (8)                     """
66: (8)                     if dict.__len__(self) == 0:
67: (12)                        self.load()
68: (12)                        setattr(self, method, types.MethodType(getattr(dict, method),
self))
69: (8)                     return getattr(dict, method)(self, *args)
70: (4)                 def __repr__(self):
71: (8)                     return self._lazy("__repr__")
72: (4)                 def __len__(self):
73: (8)                     return self._lazy("__len__")
74: (4)                 def __iter__(self):
75: (8)                     return self._lazy("__iter__")
76: (4)                 def __contains__(self, *args):
77: (8)                     return self._lazy("__contains__", *args)
78: (4)                 def __getitem__(self, *args):
79: (8)                     return self._lazy("__getitem__", *args)
80: (4)                 def __setitem__(self, *args):
81: (8)                     return self._lazy("__setitem__", *args)
82: (4)                 def setdefault(self, *args):
83: (8)                     return self._lazy("setdefault", *args)
84: (4)                 def get(self, *args, **kwargs):
85: (8)                     return self._lazy("get", *args)
86: (4)                 def items(self):
87: (8)                     return self._lazy("items")
88: (4)                 def keys(self):
89: (8)                     return self._lazy("keys")
90: (4)                 def values(self):
91: (8)                     return self._lazy("values")
92: (4)                 def update(self, *args):
93: (8)                     return self._lazy("update", *args)
94: (4)                 def pop(self, *args):
95: (8)                     return self._lazy("pop", *args)
96: (4)                 def popitem(self, *args):
97: (8)                     return self._lazy("popitem", *args)
98: (0)             class lazylist(list):
99: (4)                 def load(self):
100: (8)                    pass
101: (4)                def _lazy(self, method, *args):
102: (8)                    """If the list is empty, calls lazylist.load().
103: (8)                    Replaces lazylist.method() with list.method() and calls it.
104: (8)                    """
105: (8)                    if list.__len__(self) == 0:
106: (12)                       self.load()
107: (12)                       setattr(self, method, types.MethodType(getattr(list, method),
self))
108: (8)                    return getattr(list, method)(self, *args)
109: (4)                def __repr__(self):
110: (8)                    return self._lazy("__repr__")
111: (4)                def __len__(self):
112: (8)                    return self._lazy("__len__")
113: (4)                def __iter__(self):
114: (8)                    return self._lazy("__iter__")
115: (4)                def __contains__(self, *args):
116: (8)                    return self._lazy("__contains__", *args)
117: (4)                def insert(self, *args):
118: (8)                    return self._lazy("insert", *args)
119: (4)                def append(self, *args):
120: (8)                    return self._lazy("append", *args)
121: (4)                def extend(self, *args):
122: (8)                    return self._lazy("extend", *args)
123: (4)                def remove(self, *args):
124: (8)                    return self._lazy("remove", *args)
125: (4)                def pop(self, *args):
126: (8)                    return self._lazy("pop", *args)
127: (0)            UNIVERSAL = "universal"
128: (0)            NOUN, VERB, ADJ, ADV, PRON, DET, PREP, ADP, NUM, CONJ, INTJ, PRT, PUNC, X = (
129: (4)                "NN",
130: (4)                "VB",
```

```
131: (4)                "JJ",
132: (4)                "RB",
133: (4)                "PR",
134: (4)                "DT",
135: (4)                "PP",
136: (4)                "PP",
137: (4)                "NO",
138: (4)                "CJ",
139: (4)                "UH",
140: (4)                "PT",
141: (4)                ".",
142: (4)                "X",
143: (0)            )
144: (0)            def penntreebank2universal(token, tag):
145: (4)                """Returns a (token, tag)-tuple with a simplified universal part-of-speech
tag."""
146: (4)                if tag.startswith(("NNP-", "NNPS-")):
147: (8)                    return (token, "{}-{}".format(NOUN, tag.split("-")[-1]))
148: (4)                if tag in ("NN", "NNS", "NNP", "NNPS", "NP"):
149: (8)                    return (token, NOUN)
150: (4)                if tag in ("MD", "VB", "VBD", "VBG", "VBN", "VBP", "VBZ"):
151: (8)                    return (token, VERB)
152: (4)                if tag in ("JJ", "JJR", "JJS"):
153: (8)                    return (token, ADJ)
154: (4)                if tag in ("RB", "RBR", "RBS", "WRB"):
155: (8)                    return (token, ADV)
156: (4)                if tag in ("PRP", "PRP$", "WP", "WP$"):
157: (8)                    return (token, PRON)
158: (4)                if tag in ("DT", "PDT", "WDT", "EX"):
159: (8)                    return (token, DET)
160: (4)                if tag in ("IN",):
161: (8)                    return (token, PREP)
162: (4)                if tag in ("CD",):
163: (8)                    return (token, NUM)
164: (4)                if tag in ("CC",):
165: (8)                    return (token, CONJ)
166: (4)                if tag in ("UH",):
167: (8)                    return (token, INTJ)
168: (4)                if tag in ("POS", "RP", "TO"):
169: (8)                    return (token, PRT)
170: (4)                if tag in ("SYM", "LS", ".", "!", "?", ",", ":", "(", ")", '"', "#", "$"):
171: (8)                    return (token, PUNC)
172: (4)                return (token, X)
173: (0)            TOKEN = re.compile(r"(\S+)\s")
174: (0)            PUNCTUATION = punctuation = ".,;:!?()[]{}`''\"@#$^&*+-|=~_"
175: (0)            ABBREVIATIONS = abbreviations = set(
176: (4)                (
177: (8)                    "a.",
178: (8)                    "adj.",
179: (8)                    "adv.",
180: (8)                    "al.",
181: (8)                    "a.m.",
182: (8)                    "c.",
183: (8)                    "cf.",
184: (8)                    "comp.",
185: (8)                    "conf.",
186: (8)                    "def.",
187: (8)                    "ed.",
188: (8)                    "e.g.",
189: (8)                    "esp.",
190: (8)                    "etc.",
191: (8)                    "ex.",
192: (8)                    "f.",
193: (8)                    "fig.",
194: (8)                    "gen.",
195: (8)                    "id.",
196: (8)                    "i.e.",
197: (8)                    "int.",
198: (8)                    "l.",
```

```
199: (8)                    "m.",
200: (8)                    "Med.",
201: (8)                    "Mil.",
202: (8)                    "Mr.",
203: (8)                    "n.",
204: (8)                    "n.q.",
205: (8)                    "orig.",
206: (8)                    "pl.",
207: (8)                    "pred.",
208: (8)                    "pres.",
209: (8)                    "p.m.",
210: (8)                    "ref.",
211: (8)                    "v.",
212: (8)                    "vs.",
213: (8)                    "w/",
214: (4)                )
215: (0)            )
216: (0)            RE_ABBR1 = re.compile(r"^[A-Za-z]\.$")  # single letter, "T. De Smedt"
217: (0)            RE_ABBR2 = re.compile(r"^([A-Za-z]\.)+$")  # alternating letters, "U.S."
218: (0)            RE_ABBR3 = re.compile(
219: (4)                "^[A-Z]["
220: (4)                + "|".join(  # capital followed by consonants, "Mr."
221: (8)                    "bcdfghjklmnpqrstvwxz"
222: (4)                )
223: (4)                + "]+.$"
224: (0)            )
225: (0)            EMOTICONS = {  # (facial expression, sentiment)-keys
226: (4)                ("love", +1.00): set(("<3", "♥")),
227: (4)                ("grin", +1.00): set(
228: (8)                    (">:D", ":-D", ":D", "=-D", "=D", "X-D", "x-D", "XD", "xD", "8-D")
229: (4)                ),
230: (4)                ("taunt", +0.75): set(
231: (8)                    (">:P", ":-P", ":P", ":-p", ":p", ":-b", ":b", ":c)", ":o)", ":^)")
232: (4)                ),
233: (4)                ("smile", +0.50): set(
234: (8)                    (">:)", ":-)", ":)", "=)", "=]", ":]", ":}", ":>", ":3", "8)", "8-)")
235: (4)                ),
236: (4)                ("wink", +0.25): set((">;]", ";-)", ";)", ";-]", ";]", ";D", ";^)", "*-)",
"*)")),
237: (4)                ("gasp", +0.05): set((">:o", ":-O", ":O", ":o", ":-o", "o_O", "o.O",
"°O°", "°o°")),
238: (4)                ("worry", -0.25): set(
239: (8)                    (">:/", ":-/", ":/", ":\\", ">:\\", ":-.", ":-s", ":s", ":S", ":-S",
">.>")
240: (4)                ),
241: (4)                ("frown", -0.75): set(
242: (8)                    (">:[", ":-(", ":(", "=(", ":-[", ":[", ":{", ":-<", ":c", ":-c",
"=/")
243: (4)                ),
244: (4)                ("cry", -1.00): set((":'(", ":'''(", ";'(")),
245: (0)            }
246: (0)            RE_EMOTICONS = [
247: (4)                r" ?".join([re.escape(each) for each in e]) for v in EMOTICONS.values()
for e in v
248: (0)            ]
249: (0)            RE_EMOTICONS = re.compile(r"(%s)($|\s)" % "|".join(RE_EMOTICONS))
250: (0)            RE_SARCASM = re.compile(r"\( ?\! ?\)")
251: (0)            replacements = {
252: (4)                "'d": " 'd",
253: (4)                "'m": " 'm",
254: (4)                "'s": " 's",
255: (4)                "'ll": " 'll",
256: (4)                "'re": " 're",
257: (4)                "'ve": " 've",
258: (4)                "n't": " n't",
259: (0)            }
260: (0)            EOS = "END-OF-SENTENCE"
261: (0)            def find_tokens(
262: (4)                string,
```

```
263: (4)                punctuation=PUNCTUATION,
264: (4)                abbreviations=ABBREVIATIONS,
265: (4)                replace=replacements,
266: (4)                linebreak=r"\n{2,}",
267: (0)            ):
268: (4)                """Returns a list of sentences. Each sentence is a space-separated string
of tokens (words).
269: (4)                Handles common cases of abbreviations (e.g., etc., ...).
270: (4)                Punctuation marks are split from other words. Periods (or ?!) mark the end
of a sentence.
271: (4)                Headings without an ending period are inferred by line breaks.
272: (4)                """
273: (4)                punctuation = tuple(punctuation.replace(".", ""))
274: (4)                for a, b in list(replace.items()):
275: (8)                    string = re.sub(a, b, string)
276: (4)                if isinstance(string, str):
277: (8)                    string = (
278: (12)                       str(string)
279: (12)                       .replace("“", " “ ")
280: (12)                       .replace("”", " ” ")
281: (12)                       .replace("‘", " ‘ ")
282: (12)                       .replace("’", " ’ ")
283: (12)                       .replace("'", " ' ")
284: (12)                       .replace('"', ' " ')
285: (8)                    )
286: (4)                string = re.sub("\r\n", "\n", string)
287: (4)                string = re.sub(linebreak, " %s " % EOS, string)
288: (4)                string = re.sub(r"\s+", " ", string)
289: (4)                tokens = []
290: (4)                for t in TOKEN.findall(string + " "):
291: (8)                    if len(t) > 0:
292: (12)                       tail = []
293: (12)                       while t.startswith(punctuation) and t not in replace:
294: (16)                           if t.startswith(punctuation):
295: (20)                               tokens.append(t[0])
296: (20)                               t = t[1:]
297: (12)                       while t.endswith(punctuation + (".",)) and t not in replace:
298: (16)                           if t.endswith(punctuation):
299: (20)                               tail.append(t[-1])
300: (20)                               t = t[:-1]
301: (16)                           if t.endswith("..."):
302: (20)                               tail.append("...")
303: (20)                               t = t[:-3].rstrip(".")
304: (16)                           if t.endswith("."):
305: (20)                               if (
306: (24)                                   t in abbreviations
307: (24)                                   or RE_ABBR1.match(t) is not None
308: (24)                                   or RE_ABBR2.match(t) is not None
309: (24)                                   or RE_ABBR3.match(t) is not None
310: (20)                               ):
311: (24)                                   break
312: (20)                               else:
313: (24)                                   tail.append(t[-1])
314: (24)                                   t = t[:-1]
315: (12)                       if t != "":
316: (16)                           tokens.append(t)
317: (12)                       tokens.extend(reversed(tail))
318: (4)                sentences, i, j = [[]], 0, 0
319: (4)                while j < len(tokens):
320: (8)                    if tokens[j] in ("...", ".", "!", "?", EOS):
321: (12)                       while j < len(tokens) and tokens[j] in (
322: (16)                           "'",
323: (16)                           '"',
324: (16)                           "”",
325: (16)                           "’",
326: (16)                           "...",
327: (16)                           ".",
328: (16)                           "!",
329: (16)                           "?",
```

```
330: (16)                             ")",
331: (16)                             EOS,
332: (12)                         ):
333: (16)                             if tokens[j] in ("'", '"') and sentences[-1].count(tokens[j])
% 2 == 0:
334: (20)                                 break  # Balanced quotes.
335: (16)                             j += 1
336: (12)                         sentences[-1].extend(t for t in tokens[i:j] if t != EOS)
337: (12)                         sentences.append([])
338: (12)                         i = j
339: (8)                      j += 1
340: (4)                  sentences[-1].extend(tokens[i:j])
341: (4)                  sentences = (" ".join(s) for s in sentences if len(s) > 0)
342: (4)                  sentences = (RE_SARCASM.sub("(!)", s) for s in sentences)
343: (4)                  sentences = [
344: (8)                      RE_EMOTICONS.sub(lambda m: m.group(1).replace(" ", "") + m.group(2),
s)
345: (8)                      for s in sentences
346: (4)                  ]
347: (4)                  return sentences
348: (0)              def _read(path, encoding="utf-8", comment=";;;"):
349: (4)                  """Returns an iterator over the lines in the file at the given path,
350: (4)                  stripping comments and decoding each line to Unicode.
351: (4)                  """
352: (4)                  if path:
353: (8)                      if isinstance(path, basestring) and os.path.exists(path):
354: (12)                         f = open(path, encoding="utf-8")
355: (8)                      elif isinstance(path, basestring):
356: (12)                         f = path.splitlines()
357: (8)                      elif hasattr(path, "read"):
358: (12)                         f = path.read().splitlines()
359: (8)                      else:
360: (12)                         f = path
361: (8)                      for i, line in enumerate(f):
362: (12)                         line = (
363: (16)                             line.strip(codecs.BOM_UTF8)
364: (16)                             if i == 0 and isinstance(line, bytes)
365: (16)                             else line
366: (12)                         )
367: (12)                         line = line.strip()
368: (12)                         line = decode_utf8(line)
369: (12)                         if not line or (comment and line.startswith(comment)):
370: (16)                             continue
371: (12)                         yield line
372: (4)                  return
373: (0)              class Lexicon(lazydict):
374: (4)                  def __init__(
375: (8)                      self,
376: (8)                      path="",
377: (8)                      morphology=None,
378: (8)                      context=None,
379: (8)                      entities=None,
380: (8)                      NNP="NNP",
381: (8)                      language=None,
382: (4)                  ):
383: (8)                      """A dictionary of words and their part-of-speech tags.
384: (8)                      For unknown words, rules for word morphology, context and named
entities can be used.
385: (8)                      """
386: (8)                      self._path = path
387: (8)                      self._language = language
388: (8)                      self.morphology = Morphology(self, path=morphology)
389: (8)                      self.context = Context(self, path=context)
390: (8)                      self.entities = Entities(self, path=entities, tag=NNP)
391: (4)                  def load(self):
392: (8)                      dict.update(self, (x.split(" ")[:2] for x in _read(self._path) if
x.strip()))
393: (4)                  @property
394: (4)                  def path(self):
```

```
395: (8)                    return self._path
396: (4)                @property
397: (4)                def language(self):
398: (8)                    return self._language
399: (0)            class Rules:
400: (4)                def __init__(self, lexicon=None, cmd=None):
401: (8)                    if cmd is None:
402: (12)                       cmd = {}
403: (8)                    if lexicon is None:
404: (12)                       lexicon = {}
405: (8)                    self.lexicon, self.cmd = lexicon, cmd
406: (4)                def apply(self, x):
407: (8)                    """Applies the rule to the given token or list of tokens."""
408: (8)                    return x
409: (0)            class Morphology(lazylist, Rules):
410: (4)                def __init__(self, lexicon=None, path=""):
411: (8)                    """A list of rules based on word morphology (prefix, suffix)."""
412: (8)                    if lexicon is None:
413: (12)                       lexicon = {}
414: (8)                    cmd = (
415: (12)                       "char",  # Word contains x.
416: (12)                       "haspref",  # Word starts with x.
417: (12)                       "hassuf",  # Word end with x.
418: (12)                       "addpref",  # x + word is in lexicon.
419: (12)                       "addsuf",  # Word + x is in lexicon.
420: (12)                       "deletepref",  # Word without x at the start is in lexicon.
421: (12)                       "deletesuf",  # Word without x at the end is in lexicon.
422: (12)                       "goodleft",  # Word preceded by word x.
423: (12)                       "goodright",  # Word followed by word x.
424: (8)                    )
425: (8)                    cmd = dict.fromkeys(cmd, True)
426: (8)                    cmd.update(("f" + k, v) for k, v in list(cmd.items()))
427: (8)                    Rules.__init__(self, lexicon, cmd)
428: (8)                    self._path = path
429: (4)                @property
430: (4)                def path(self):
431: (8)                    return self._path
432: (4)                def load(self):
433: (8)                    list.extend(self, (x.split() for x in _read(self._path)))
434: (4)                def apply(self, token, previous=(None, None), next=(None, None)):
435: (8)                    """Applies lexical rules to the given token, which is a [word, tag]
list."""
436: (8)                    w = token[0]
437: (8)                    for r in self:
438: (12)                       if r[1] in self.cmd:  # Rule = ly hassuf 2 RB x
439: (16)                           f, x, pos, cmd = bool(0), r[0], r[-2], r[1].lower()
440: (12)                       if r[2] in self.cmd:  # Rule = NN s fhassuf 1 NNS x
441: (16)                           f, x, pos, cmd = bool(1), r[1], r[-2],
r[2].lower().lstrip("f")
442: (12)                       if f and token[1] != r[0]:
443: (16)                           continue
444: (12)                       if (
445: (16)                           (cmd == "char" and x in w)
446: (16)                           or (cmd == "haspref" and w.startswith(x))
447: (16)                           or (cmd == "hassuf" and w.endswith(x))
448: (16)                           or (cmd == "addpref" and x + w in self.lexicon)
449: (16)                           or (cmd == "addsuf" and w + x in self.lexicon)
450: (16)                           or (
451: (20)                               cmd == "deletepref"
452: (20)                               and w.startswith(x)
453: (20)                               and w[len(x) :] in self.lexicon
454: (16)                           )
455: (16)                           or (
456: (20)                               cmd == "deletesuf"
457: (20)                               and w.endswith(x)
458: (20)                               and w[: -len(x)] in self.lexicon
459: (16)                           )
460: (16)                           or (cmd == "goodleft" and x == next[0])
461: (16)                           or (cmd == "goodright" and x == previous[0])
```

```
462: (12)                         ):
463: (16)                             token[1] = pos
464: (8)                     return token
465: (4)                 def insert(self, i, tag, affix, cmd="hassuf", tagged=None):
466: (8)                     """Inserts a new rule that assigns the given tag to words with the
given affix,
467: (8)                     e.g., Morphology.append("RB", "-ly").
468: (8)                     """
469: (8)                     if affix.startswith("-") and affix.endswith("-"):
470: (12)                        affix, cmd = affix[+1:-1], "char"
471: (8)                     if affix.startswith("-"):
472: (12)                        affix, cmd = affix[+1:-0], "hassuf"
473: (8)                     if affix.endswith("-"):
474: (12)                        affix, cmd = affix[+0:-1], "haspref"
475: (8)                     if tagged:
476: (12)                        r = [tagged, affix, "f" + cmd.lstrip("f"), tag, "x"]
477: (8)                     else:
478: (12)                        r = [affix, cmd.lstrip("f"), tag, "x"]
479: (8)                     lazylist.insert(self, i, r)
480: (4)                 def append(self, *args, **kwargs):
481: (8)                     self.insert(len(self) - 1, *args, **kwargs)
482: (4)                 def extend(self, rules=None):
483: (8)                     if rules is None:
484: (12)                        rules = []
485: (8)                     for r in rules:
486: (12)                        self.append(*r)
487: (0)             class Context(lazylist, Rules):
488: (4)                 def __init__(self, lexicon=None, path=""):
489: (8)                     """A list of rules based on context (preceding and following
words)."""
490: (8)                     if lexicon is None:
491: (12)                        lexicon = {}
492: (8)                     cmd = (
493: (12)                        "prevtag",  # Preceding word is tagged x.
494: (12)                        "nexttag",  # Following word is tagged x.
495: (12)                        "prev2tag",  # Word 2 before is tagged x.
496: (12)                        "next2tag",  # Word 2 after is tagged x.
497: (12)                        "prev1or2tag",  # One of 2 preceding words is tagged x.
498: (12)                        "next1or2tag",  # One of 2 following words is tagged x.
499: (12)                        "prev1or2or3tag",  # One of 3 preceding words is tagged x.
500: (12)                        "next1or2or3tag",  # One of 3 following words is tagged x.
501: (12)                        "surroundtag",  # Preceding word is tagged x and following word is
tagged y.
502: (12)                        "curwd",  # Current word is x.
503: (12)                        "prevwd",  # Preceding word is x.
504: (12)                        "nextwd",  # Following word is x.
505: (12)                        "prev1or2wd",  # One of 2 preceding words is x.
506: (12)                        "next1or2wd",  # One of 2 following words is x.
507: (12)                        "next1or2or3wd",  # One of 3 preceding words is x.
508: (12)                        "prev1or2or3wd",  # One of 3 following words is x.
509: (12)                        "prevwdtag",  # Preceding word is x and tagged y.
510: (12)                        "nextwdtag",  # Following word is x and tagged y.
511: (12)                        "wdprevtag",  # Current word is y and preceding word is tagged x.
512: (12)                        "wdnexttag",  # Current word is x and following word is tagged y.
513: (12)                        "wdand2aft",  # Current word is x and word 2 after is y.
514: (12)                        "wdand2tagbfr",  # Current word is y and word 2 before is tagged
x.
515: (12)                        "wdand2tagaft",  # Current word is x and word 2 after is tagged y.
516: (12)                        "lbigram",  # Current word is y and word before is x.
517: (12)                        "rbigram",  # Current word is x and word after is y.
518: (12)                        "prevbigram",  # Preceding word is tagged x and word before is
tagged y.
519: (12)                        "nextbigram",  # Following word is tagged x and word after is
tagged y.
520: (8)                     )
521: (8)                     Rules.__init__(self, lexicon, dict.fromkeys(cmd, True))
522: (8)                     self._path = path
523: (4)                 @property
524: (4)                 def path(self):
```

```
525: (8)                    return self._path
526: (4)                def load(self):
527: (8)                    list.extend(self, (x.split() for x in _read(self._path)))
528: (4)                def apply(self, tokens):
529: (8)                    """Applies contextual rules to the given list of tokens,
530: (8)                    where each token is a [word, tag] list.
531: (8)                    """
532: (8)                    o = [("STAART", "STAART")] * 3  # Empty delimiters for look
ahead/back.
533: (8)                    t = o + tokens + o
534: (8)                    for i, token in enumerate(t):
535: (12)                       for r in self:
536: (16)                           if token[1] == "STAART":
537: (20)                               continue
538: (16)                           if token[1] != r[0] and r[0] != "*":
539: (20)                               continue
540: (16)                           cmd, x, y = r[2], r[3], r[4] if len(r) > 4 else ""
541: (16)                           cmd = cmd.lower()
542: (16)                           if (
543: (20)                               (cmd == "prevtag" and x == t[i - 1][1])
544: (20)                               or (cmd == "nexttag" and x == t[i + 1][1])
545: (20)                               or (cmd == "prev2tag" and x == t[i - 2][1])
546: (20)                               or (cmd == "next2tag" and x == t[i + 2][1])
547: (20)                               or (cmd == "prev1or2tag" and x in (t[i - 1][1], t[i - 2]
[1]))
548: (20)                               or (cmd == "next1or2tag" and x in (t[i + 1][1], t[i + 2]
[1]))
549: (20)                               or (
550: (24)                                   cmd == "prev1or2or3tag"
551: (24)                                   and x in (t[i - 1][1], t[i - 2][1], t[i - 3][1])
552: (20)                               )
553: (20)                               or (
554: (24)                                   cmd == "next1or2or3tag"
555: (24)                                   and x in (t[i + 1][1], t[i + 2][1], t[i + 3][1])
556: (20)                               )
557: (20)                               or (cmd == "surroundtag" and x == t[i - 1][1] and y == t[i
+ 1][1])
558: (20)                               or (cmd == "curwd" and x == t[i + 0][0])
559: (20)                               or (cmd == "prevwd" and x == t[i - 1][0])
560: (20)                               or (cmd == "nextwd" and x == t[i + 1][0])
561: (20)                               or (cmd == "prev1or2wd" and x in (t[i - 1][0], t[i - 2]
[0]))
562: (20)                               or (cmd == "next1or2wd" and x in (t[i + 1][0], t[i + 2]
[0]))
563: (20)                               or (cmd == "prevwdtag" and x == t[i - 1][0] and y == t[i -
1][1])
564: (20)                               or (cmd == "nextwdtag" and x == t[i + 1][0] and y == t[i +
1][1])
565: (20)                               or (cmd == "wdprevtag" and x == t[i - 1][1] and y == t[i +
0][0])
566: (20)                               or (cmd == "wdnexttag" and x == t[i + 0][0] and y == t[i +
1][1])
567: (20)                               or (cmd == "wdand2aft" and x == t[i + 0][0] and y == t[i +
2][0])
568: (20)                               or (cmd == "wdand2tagbfr" and x == t[i - 2][1] and y ==
t[i + 0][0])
569: (20)                               or (cmd == "wdand2tagaft" and x == t[i + 0][0] and y ==
t[i + 2][1])
570: (20)                               or (cmd == "lbigram" and x == t[i - 1][0] and y == t[i +
0][0])
571: (20)                               or (cmd == "rbigram" and x == t[i + 0][0] and y == t[i +
1][0])
572: (20)                               or (cmd == "prevbigram" and x == t[i - 2][1] and y == t[i
- 1][1])
573: (20)                               or (cmd == "nextbigram" and x == t[i + 1][1] and y == t[i
+ 2][1])
574: (16)                           ):
575: (20)                               t[i] = [t[i][0], r[1]]
576: (8)                    return t[len(o) : -len(o)]
```

```
577: (4)                def insert(self, i, tag1, tag2, cmd="prevtag", x=None, y=None):
578: (8)                    """Inserts a new rule that updates words with tag1 to tag2,
579: (8)                    given constraints x and y, e.g., Context.append("TO < NN", "VB")
580: (8)                    """
581: (8)                    if " < " in tag1 and not x and not y:
582: (12)                       tag1, x = tag1.split(" < ")
583: (12)                       cmd = "prevtag"
584: (8)                    if " > " in tag1 and not x and not y:
585: (12)                       x, tag1 = tag1.split(" > ")
586: (12)                       cmd = "nexttag"
587: (8)                    lazylist.insert(self, i, [tag1, tag2, cmd, x or "", y or ""])
588: (4)                def append(self, *args, **kwargs):
589: (8)                    self.insert(len(self) - 1, *args, **kwargs)
590: (4)                def extend(self, rules=None):
591: (8)                    if rules is None:
592: (12)                       rules = []
593: (8)                    for r in rules:
594: (12)                       self.append(*r)
595: (0)            RE_ENTITY1 = re.compile(r"^http://")  # http://www.domain.com/path
596: (0)            RE_ENTITY2 = re.compile(r"^www\..*?\.[com|org|net|edu|de|uk]$")  #
www.domain.com
597: (0)            RE_ENTITY3 = re.compile(r"^[\w\-\.\+]+@(\w[\w\-]+\.)+[\w\-]+$")  #
name@domain.com
598: (0)            class Entities(lazydict, Rules):
599: (4)                def __init__(self, lexicon=None, path="", tag="NNP"):
600: (8)                    """A dictionary of named entities and their labels.
601: (8)                    For domain names and e-mail adresses, regular expressions are used.
602: (8)                    """
603: (8)                    if lexicon is None:
604: (12)                       lexicon = {}
605: (8)                    cmd = (
606: (12)                       "pers",  # Persons: George/NNP-PERS
607: (12)                       "loc",  # Locations: Washington/NNP-LOC
608: (12)                       "org",  # Organizations: Google/NNP-ORG
609: (8)                    )
610: (8)                    Rules.__init__(self, lexicon, cmd)
611: (8)                    self._path = path
612: (8)                    self.tag = tag
613: (4)                @property
614: (4)                def path(self):
615: (8)                    return self._path
616: (4)                def load(self):
617: (8)                    for x in _read(self.path):
618: (12)                       x = [x.lower() for x in x.split()]
619: (12)                       dict.setdefault(self, x[0], []).append(x)
620: (4)                def apply(self, tokens):
621: (8)                    """Applies the named entity recognizer to the given list of tokens,
622: (8)                    where each token is a [word, tag] list.
623: (8)                    """
624: (8)                    i = 0
625: (8)                    while i < len(tokens):
626: (12)                       w = tokens[i][0].lower()
627: (12)                       if RE_ENTITY1.match(w) or RE_ENTITY2.match(w) or
RE_ENTITY3.match(w):
628: (16)                           tokens[i][1] = self.tag
629: (12)                       if w in self:
630: (16)                           for e in self[w]:
631: (20)                               e, tag = (
632: (24)                                   (e[:-1], "-" + e[-1].upper()) if e[-1] in self.cmd
else (e, "")
633: (20)                               )
634: (20)                               b = True
635: (20)                               for j, e in enumerate(e):
636: (24)                                   if i + j >= len(tokens) or tokens[i + j][0].lower() !=
e:
637: (28)                                       b = False
638: (28)                                       break
639: (20)                               if b:
640: (24)                                   for token in tokens[i : i + j + 1]:
```

```
641: (28)                                    token[1] = (
642: (32)                                        token[1] == "NNPS" and token[1] or self.tag
643: (28)                                    ) + tag
644: (24)                                i += j
645: (24)                                break
646: (12)                    i += 1
647: (8)                return tokens
648: (4)            def append(self, entity, name="pers"):
649: (8)                """Appends a named entity to the lexicon,
650: (8)                e.g., Entities.append("Hooloovoo", "PERS")
651: (8)                """
652: (8)                e = [s.lower() for s in entity.split(" ") + [name]]
653: (8)                self.setdefault(e[0], []).append(e)
654: (4)            def extend(self, entities):
655: (8)                for entity, name in entities:
656: (12)                   self.append(entity, name)
657: (0)        MOOD = "mood"  # emoticons, emojis
658: (0)        IRONY = "irony"  # sarcasm mark (!)
659: (0)        NOUN, VERB, ADJECTIVE, ADVERB = "NN", "VB", "JJ", "RB"
660: (0)        RE_SYNSET = re.compile(r"^[acdnrv][-_][0-9]+$")
661: (0)        def avg(list):
662: (4)            return sum(list) / float(len(list) or 1)
663: (0)        class Score(tuple):
664: (4)            def __new__(self, polarity, subjectivity, assessments=None):
665: (8)                """A (polarity, subjectivity)-tuple with an assessments property."""
666: (8)                if assessments is None:
667: (12)                   assessments = []
668: (8)                return tuple.__new__(self, [polarity, subjectivity])
669: (4)            def __init__(self, polarity, subjectivity, assessments=None):
670: (8)                if assessments is None:
671: (12)                   assessments = []
672: (8)                self.assessments = assessments
673: (0)        class Sentiment(lazydict):
674: (4)            def __init__(self, path="", language=None, synset=None, confidence=None,
**kwargs):
675: (8)                """A dictionary of words (adjectives) and polarity scores
(positive/negative).
676: (8)                The value for each word is a dictionary of part-of-speech tags.
677: (8)                The value for each word POS-tag is a tuple with values for
678: (8)                polarity (-1.0-1.0), subjectivity (0.0-1.0) and intensity (0.5-2.0).
679: (8)                """
680: (8)                self._path = path  # XML file path.
681: (8)                self._language = None  # XML language attribute ("en", "fr", ...)
682: (8)                self._confidence = None  # XML confidence attribute threshold (>=).
683: (8)                self._synset = synset  # XML synset attribute ("wordnet_id",
"cornetto_id", ...)
684: (8)                self._synsets = {}  # {"a-01123879": (1.0, 1.0, 1.0)}
685: (8)                self.labeler = {}  # {"dammit": "profanity"}
686: (8)                self.tokenizer = kwargs.get("tokenizer", find_tokens)
687: (8)                self.negations = kwargs.get("negations", ("no", "not", "n't",
"never"))
688: (8)                self.modifiers = kwargs.get("modifiers", ("RB",))
689: (8)                self.modifier = kwargs.get("modifier", lambda w: w.endswith("ly"))
690: (4)            @property
691: (4)            def path(self):
692: (8)                return self._path
693: (4)            @property
694: (4)            def language(self):
695: (8)                return self._language
696: (4)            @property
697: (4)            def confidence(self):
698: (8)                return self._confidence
699: (4)            def load(self, path=None):
700: (8)                """Loads the XML-file (with sentiment annotations) from the given
path.
701: (8)                By default, Sentiment.path is lazily loaded.
702: (8)                """
703: (8)                if not path:
704: (12)                   path = self._path
```

```
                    if not os.path.exists(path):
                        return
                    words, synsets, labels = {}, {}, {}
                    xml = ElementTree.parse(path)
                    xml = xml.getroot()
                    for w in xml.findall("word"):
                        if self._confidence is None or self._confidence <= float(
                            w.attrib.get("confidence", 0.0)
                        ):
                            w, pos, p, s, i, label, synset = (
                                w.attrib.get("form"),
                                w.attrib.get("pos"),
                                w.attrib.get("polarity", 0.0),
                                w.attrib.get("subjectivity", 0.0),
                                w.attrib.get("intensity", 1.0),
                                w.attrib.get("label"),
                                w.attrib.get(self._synset),  # wordnet_id, cornetto_id,
...
                            )
                            psi = (float(p), float(s), float(i))
                            if w:
                                words.setdefault(w, {}).setdefault(pos, []).append(psi)
                            if w and label:
                                labels[w] = label
                            if synset:
                                synsets.setdefault(synset, []).append(psi)
                    self._language = xml.attrib.get("language", self._language)
                    for w in words:
                        words[w] = dict(
                            (pos, [avg(each) for each in zip(*psi)])
                            for pos, psi in words[w].items()
                        )
                    for w, pos in list(words.items()):
                        words[w][None] = [avg(each) for each in zip(*pos.values())]
                    for id, psi in synsets.items():
                        synsets[id] = [avg(each) for each in zip(*psi)]
                    dict.update(self, words)
                    dict.update(self.labeler, labels)
                    dict.update(self._synsets, synsets)
                def synset(self, id, pos=ADJECTIVE):
                    """Returns a (polarity, subjectivity)-tuple for the given synset id.
                    For example, the adjective "horrible" has id 193480 in WordNet:
                    Sentiment.synset(193480, pos="JJ") => (-0.6, 1.0, 1.0).
                    """
                    id = str(id).zfill(8)
                    if not id.startswith(("n-", "v-", "a-", "r-")):
                        if pos == NOUN:
                            id = "n-" + id
                        if pos == VERB:
                            id = "v-" + id
                        if pos == ADJECTIVE:
                            id = "a-" + id
                        if pos == ADVERB:
                            id = "r-" + id
                    if dict.__len__(self) == 0:
                        self.load()
                    return tuple(self._synsets.get(id, (0.0, 0.0))[:2])
                def __call__(self, s, negation=True, **kwargs):
                    """Returns a (polarity, subjectivity)-tuple for the given sentence,
                    with polarity between -1.0 and 1.0 and subjectivity between 0.0 and
1.0.
                    The sentence can be a string, Synset, Text, Sentence, Chunk, Word,
Document, Vector.
                    An optional weight parameter can be given,
                    as a function that takes a list of words and returns a weight.
                    """
                    def avg(assessments, weighted=lambda w: 1):
                        s, n = 0, 0
                        for words, score in assessments:
```

```
771: (16)                             w = weighted(words)
772: (16)                             s += w * score
773: (16)                             n += w
774: (12)                         return s / float(n or 1)
775: (8)                      if hasattr(s, "gloss"):
776: (12)                         a = [(s.synonyms[0],) + self.synset(s.id, pos=s.pos) + (None,)]
777: (8)                      elif (
778: (12)                         isinstance(s, basestring) and RE_SYNSET.match(s) and hasattr(s,
"synonyms")
779: (8)                      ):
780: (12)                         a = [(s.synonyms[0],) + self.synset(s.id, pos=s.pos) + (None,)]
781: (8)                      elif isinstance(s, basestring):
782: (12)                         a = self.assessments(
783: (16)                             ((w.lower(), None) for w in "
".join(self.tokenizer(s)).split()),
784: (16)                             negation,
785: (12)                         )
786: (8)                      elif hasattr(s, "sentences"):
787: (12)                         a = self.assessments(
788: (16)                             (
789: (20)                                 (w.lemma or w.string.lower(), w.pos[:2])
790: (20)                                 for w in chain.from_iterable(s)
791: (16)                             ),
792: (16)                             negation,
793: (12)                         )
794: (8)                      elif hasattr(s, "lemmata"):
795: (12)                         a = self.assessments(
796: (16)                             ((w.lemma or w.string.lower(), w.pos[:2]) for w in s.words),
negation
797: (12)                         )
798: (8)                      elif hasattr(s, "lemma"):
799: (12)                         a = self.assessments(((s.lemma or s.string.lower(), s.pos[:2]),),
negation)
800: (8)                      elif hasattr(s, "terms"):
801: (12)                         a = self.assessments(
802: (16)                             chain.from_iterable(((w, None), (None, None)) for w in s),
negation
803: (12)                         )
804: (12)                         kwargs.setdefault("weight", lambda w: s.terms[w[0]])
805: (8)                      elif isinstance(s, dict):
806: (12)                         a = self.assessments(
807: (16)                             chain.from_iterable(((w, None), (None, None)) for w in s),
negation
808: (12)                         )
809: (12)                         kwargs.setdefault("weight", lambda w: s[w[0]])
810: (8)                      elif isinstance(s, list):
811: (12)                         a = self.assessments(((w, None) for w in s), negation)
812: (8)                      else:
813: (12)                         a = []
814: (8)                      weight = kwargs.get("weight", lambda w: 1)  # [(w, p) for w, p, s, x
in a]
815: (8)                      return Score(
816: (12)                         polarity=avg([(w, p) for w, p, s, x in a], weight),
817: (12)                         subjectivity=avg([(w, s) for w, p, s, x in a], weight),
818: (12)                         assessments=a,
819: (8)                      )
820: (4)                  def assessments(self, words=None, negation=True):
821: (8)                      """Returns a list of (chunk, polarity, subjectivity, label)-tuples for
the given list of words:
822: (8)                      where chunk is a list of successive words: a known word optionally
823: (8)                      preceded by a modifier ("very good") or a negation ("not good").
824: (8)                      """
825: (8)                      if words is None:
826: (12)                         words = []
827: (8)                      a = []
828: (8)                      m = None  # Preceding modifier (i.e., adverb or adjective).
829: (8)                      n = None  # Preceding negation (e.g., "not beautiful").
830: (8)                      for w, pos in words:
831: (12)                         if w is None:
```

```
                            continue
                        if w in self and pos in self[w]:
                            p, s, i = self[w][pos]
                            if m is None:
                                a.append(dict(w=[w], p=p, s=s, i=i, n=1,
x=self.labeler.get(w)))
                            if m is not None:
                                a[-1]["w"].append(w)
                                a[-1]["p"] = max(-1.0, min(p * a[-1]["i"], +1.0))
                                a[-1]["s"] = max(-1.0, min(s * a[-1]["i"], +1.0))
                                a[-1]["i"] = i
                                a[-1]["x"] = self.labeler.get(w)
                            if n is not None:
                                a[-1]["w"].insert(0, n)
                                a[-1]["i"] = 1.0 / a[-1]["i"]
                                a[-1]["n"] = -1
                            m = None
                            n = None
                            if (
                                pos
                                and pos in self.modifiers
                                or any(map(self[w].__contains__, self.modifiers))
                            ):
                                m = (w, pos)
                            if negation and w in self.negations:
                                n = w
                        else:
                            if negation and w in self.negations:
                                n = w
                            elif n and len(w.strip("'")) > 1:
                                n = None
                            if (
                                n is not None
                                and m is not None
                                and (pos in self.modifiers or self.modifier(m[0]))
                            ):
                                a[-1]["w"].append(n)
                                a[-1]["n"] = -1
                                n = None
                            elif m and len(w) > 2:
                                m = None
                            if w == "!" and len(a) > 0:
                                a[-1]["w"].append("!")
                                a[-1]["p"] = max(-1.0, min(a[-1]["p"] * 1.25, +1.0))
                            if w == "(!)":
                                a.append(dict(w=[w], p=0.0, s=1.0, i=1.0, n=1, x=IRONY))
                            if (
                                w.isalpha() is False and len(w) <= 5 and w not in
PUNCTUATION
                            ):  # speedup
                                for (_type, p), e in EMOTICONS.items():
                                    if w in map(lambda e: e.lower(), e):
                                        a.append(dict(w=[w], p=p, s=1.0, i=1.0, n=1,
x=MOOD))
                                        break
                    for i in range(len(a)):
                        w = a[i]["w"]
                        p = a[i]["p"]
                        s = a[i]["s"]
                        n = a[i]["n"]
                        x = a[i]["x"]
                        a[i] = (w, p * -0.5 if n < 0 else p, s, x)
                    return a
                def annotate(
                    self, word, pos=None, polarity=0.0, subjectivity=0.0, intensity=1.0,
label=None
                ):
                    """Annotates the given word with polarity, subjectivity and intensity
scores,
```

```
896: (8)                    and optionally a semantic label (e.g., MOOD for emoticons, IRONY for "
(!)").
897: (8)                    """
898: (8)                    w = self.setdefault(word, {})
899: (8)                    w[pos] = w[None] = (polarity, subjectivity, intensity)
900: (8)                    if label:
901: (12)                       self.labeler[word] = label
902: (0)            CD = re.compile(r"^[0-9\-\,\.\:\/\%\$]+$")
903: (0)            def _suffix_rules(token, tag="NN"):
904: (4)                """Default morphological tagging rules for English, based on word
suffixes."""
905: (4)                if isinstance(token, (list, tuple)):
906: (8)                    token, tag = token
907: (4)                if token.endswith("ing"):
908: (8)                    tag = "VBG"
909: (4)                if token.endswith("ly"):
910: (8)                    tag = "RB"
911: (4)                if token.endswith("s") and not token.endswith(("is", "ous", "ss")):
912: (8)                    tag = "NNS"
913: (4)                if (
914: (8)                    token.endswith(
915: (12)                       ("able", "al", "ful", "ible", "ient", "ish", "ive", "less", "tic",
"ous")
916: (8)                    )
917: (8)                    or "-" in token
918: (4)                ):
919: (8)                    tag = "JJ"
920: (4)                if token.endswith("ed"):
921: (8)                    tag = "VBN"
922: (4)                if token.endswith(("ate", "ify", "ise", "ize")):
923: (8)                    tag = "VBP"
924: (4)                return [token, tag]
925: (0)            def find_tags(
926: (4)                tokens,
927: (4)                lexicon=None,
928: (4)                model=None,
929: (4)                morphology=None,
930: (4)                context=None,
931: (4)                entities=None,
932: (4)                default=("NN", "NNP", "CD"),
933: (4)                language="en",
934: (4)                map=None,
935: (4)                **kwargs,
936: (0)            ):
937: (4)                """Returns a list of [token, tag]-items for the given list of tokens:
938: (4)                ["The", "cat", "purs"] => [["The", "DT"], ["cat", "NN"], ["purs", "VB"]]
939: (4)                Words are tagged using the given lexicon of (word, tag)-items.
940: (4)                Unknown words are tagged NN by default.
941: (4)                Unknown words that start with a capital letter are tagged NNP (unless
language="de").
942: (4)                Unknown words that consist only of digits and punctuation marks are tagged
CD.
943: (4)                Unknown words are then improved with morphological rules.
944: (4)                All words are improved with contextual rules.
945: (4)                If a model is given, uses model for unknown words instead of morphology
and context.
946: (4)                If map is a function, it is applied to each (token, tag) after applying
all rules.
947: (4)                """
948: (4)                if lexicon is None:
949: (8)                    lexicon = {}
950: (4)                tagged = []
951: (4)                for i, token in enumerate(tokens):
952: (8)                    tagged.append(
953: (12)                       [token, lexicon.get(token, i == 0 and lexicon.get(token.lower())
or None)]
954: (8)                    )
955: (4)                for i, (token, tag) in enumerate(tagged):
956: (8)                    prev, next = (None, None), (None, None)
```

```
957: (8)                    if i > 0:
958: (12)                       prev = tagged[i - 1]
959: (8)                    if i < len(tagged) - 1:
960: (12)                       next = tagged[i + 1]
961: (8)                    if tag is None or token in (model is not None and model.unknown or
()):
962: (12)                       if model is not None:
963: (16)                           tagged[i] = model.apply([token, None], prev, next)
964: (12)                       elif token.istitle() and language != "de":
965: (16)                           tagged[i] = [token, default[1]]
966: (12)                       elif CD.match(token) is not None:
967: (16)                           tagged[i] = [token, default[2]]
968: (12)                       elif morphology is not None:
969: (16)                           tagged[i] = morphology.apply([token, default[0]], prev, next)
970: (12)                       elif language == "en":
971: (16)                           tagged[i] = _suffix_rules([token, default[0]])
972: (12)                       else:
973: (16)                           tagged[i] = [token, default[0]]
974: (4)                if context is not None and model is None:
975: (8)                    tagged = context.apply(tagged)
976: (4)                if entities is not None:
977: (8)                    tagged = entities.apply(tagged)
978: (4)                if map is not None:
979: (8)                    tagged = [list(map(token, tag)) or [token, default[0]] for token, tag
in tagged]
980: (4)                return tagged
981: (0)            SEPARATOR = "/"
982: (0)            NN = r"NN|NNS|NNP|NNPS|NNPS?\-[A-Z]{3,4}|PR|PRP|PRP\$"
983: (0)            VB = r"VB|VBD|VBG|VBN|VBP|VBZ"
984: (0)            JJ = r"JJ|JJR|JJS"
985: (0)            RB = r"(?<!W)RB|RBR|RBS"
986: (0)            CHUNKS = [
987: (4)                [
988: (8)                    (
989: (12)                       "NP",
990: (12)                       re.compile(
991: (16)                           r"(("
992: (16)                           + NN
993: (16)                           + ")/)*((DT|CD|CC|CJ)/)*(("
994: (16)                           + RB
995: (16)                           + "|"
996: (16)                           + JJ
997: (16)                           + ")/)*(("
998: (16)                           + NN
999: (16)                           + ")/)+"
1000: (12)                      ),
1001: (8)                   ),
1002: (8)                   ("VP", re.compile(r"(((MD|" + RB + ")/)*((" + VB + ")/)+)+")),
1003: (8)                   ("VP", re.compile(r"((MD)/)")),
1004: (8)                   ("PP", re.compile(r"((IN|PP|TO)/)+")),
1005: (8)                   ("ADJP", re.compile(r"((CC|CJ|" + RB + "|" + JJ + ")/)*((" + JJ +
")/)+")),
1006: (8)                   ("ADVP", re.compile(r"((" + RB + "|WRB)/)+")),
1007: (4)               ],
1008: (4)               [
1009: (8)                   (
1010: (12)                      "NP",
1011: (12)                      re.compile(
1012: (16)                          r"(("
1013: (16)                          + NN
1014: (16)                          + ")/)*((DT|CD|CC|CJ)/)*(("
1015: (16)                          + RB
1016: (16)                          + "|"
1017: (16)                          + JJ
1018: (16)                          + ")/)*(("
1019: (16)                          + NN
1020: (16)                          + ")/)+(("
1021: (16)                          + RB
1022: (16)                          + "|"
```

```
                        + JJ
                        + ")/)*"
                    ),
                ),
                ("VP", re.compile(r"(((MD|" + RB + ")/)*((" + VB + ")/)+((" + RB +
")/)*)+")),
                ("VP", re.compile(r"((MD)/)")),
                ("PP", re.compile(r"((IN|PP|TO)/)+")),
                ("ADJP", re.compile(r"((CC|CJ|" + RB + "|" + JJ + ")/)*((" + JJ +
")/)+")),
                ("ADVP", re.compile(r"((" + RB + "|WRB)/)+")),
            ],
        ]
        CHUNKS[0].insert(1, CHUNKS[0].pop(3))
        CHUNKS[1].insert(1, CHUNKS[1].pop(3))
        def find_chunks(tagged, language="en"):
            """The input is a list of [token, tag]-items.
            The output is a list of [token, tag, chunk]-items:
            The/DT nice/JJ fish/NN is/VBZ dead/JJ ./. =>
            The/DT/B-NP nice/JJ/I-NP fish/NN/I-NP is/VBZ/B-VP dead/JJ/B-ADJP ././O
            """
            chunked = [x for x in tagged]
            tags = "".join(f"{tag}{SEPARATOR}" for token, tag in tagged)
            for tag, rule in CHUNKS[
                int(language in ("ca", "es", "pt", "fr", "it", "pt", "ro"))
            ]:
                for m in rule.finditer(tags):
                    i = m.start()
                    j = tags[:i].count(SEPARATOR)
                    n = m.group(0).count(SEPARATOR)
                    for k in range(j, j + n):
                        if len(chunked[k]) == 3:
                            continue
                        if len(chunked[k]) < 3:
                            if k == j and chunked[k][1] in ("CC", "CJ", "KON",
"Conj(neven)"):
                                j += 1
                            elif k == j:
                                chunked[k].append("B-" + tag)
                            else:
                                chunked[k].append("I-" + tag)
            for chink in filter(lambda x: len(x) < 3, chunked):
                chink.append("O")
            for i, (_word, tag, chunk) in enumerate(chunked):
                if tag.startswith("RB") and chunk == "B-NP":
                    if i < len(chunked) - 1 and not chunked[i + 1]
[1].startswith("JJ"):
                        chunked[i + 0][2] = "B-ADVP"
                        chunked[i + 1][2] = "B-NP"
            return chunked
        def find_prepositions(chunked):
            """The input is a list of [token, tag, chunk]-items.
            The output is a list of [token, tag, chunk, preposition]-items.
            PP-chunks followed by NP-chunks make up a PNP-chunk.
            """
            for ch in chunked:
                ch.append("O")
            for i, chunk in enumerate(chunked):
                if chunk[2].endswith("PP") and chunk[-1] == "O":
                    if i < len(chunked) - 1 and (
                        chunked[i + 1][2].endswith(("NP", "PP"))
                        or chunked[i + 1][1] in ("VBG", "VBN")
                    ):
                        chunk[-1] = "B-PNP"
                        pp = True
                        for ch in chunked[i + 1 :]:
                            if not (ch[2].endswith(("NP", "PP")) or ch[1] in ("VBG",
"VBN")):
                                break
```

```
1087: (20)                             if ch[2].endswith("PP") and pp:
1088: (24)                                 ch[-1] = "I-PNP"
1089: (20)                             if not ch[2].endswith("PP"):
1090: (24)                                 ch[-1] = "I-PNP"
1091: (24)                                 pp = False
1092: (4)             return chunked
1093: (0)         PTB = PENN = "penn"
1094: (0)         class Parser:
1095: (4)             def __init__(self, lexicon=None, default=("NN", "NNP", "CD"),
language=None):
1096: (8)                 """A simple shallow parser using a Brill-based part-of-speech tagger.
1097: (8)                 The given lexicon is a dictionary of known words and their part-of-
speech tag.
1098: (8)                 The given default tags are used for unknown words.
1099: (8)                 Unknown words that start with a capital letter are tagged NNP (except
for German).
1100: (8)                 Unknown words that contain only digits and punctuation are tagged CD.
1101: (8)                 The given language can be used to discern between
1102: (8)                 Germanic and Romance languages for phrase chunking.
1103: (8)                 """
1104: (8)                 if lexicon is None:
1105: (12)                    lexicon = {}
1106: (8)                 self.lexicon = lexicon
1107: (8)                 self.default = default
1108: (8)                 self.language = language
1109: (4)             def find_tokens(self, string, **kwargs):
1110: (8)                 """Returns a list of sentences from the given string.
1111: (8)                 Punctuation marks are separated from each word by a space.
1112: (8)                 """
1113: (8)                 return find_tokens(
1114: (12)                    str(string),
1115: (12)                    punctuation=kwargs.get("punctuation", PUNCTUATION),
1116: (12)                    abbreviations=kwargs.get("abbreviations", ABBREVIATIONS),
1117: (12)                    replace=kwargs.get("replace", replacements),
1118: (12)                    linebreak=r"\n{2,}",
1119: (8)                 )
1120: (4)             def find_tags(self, tokens, **kwargs):
1121: (8)                 """Annotates the given list of tokens with part-of-speech tags.
1122: (8)                 Returns a list of tokens, where each token is now a [word, tag]-list.
1123: (8)                 """
1124: (8)                 return find_tags(
1125: (12)                    tokens,
1126: (12)                    language=kwargs.get("language", self.language),
1127: (12)                    lexicon=kwargs.get("lexicon", self.lexicon),
1128: (12)                    default=kwargs.get("default", self.default),
1129: (12)                    map=kwargs.get("map", None),
1130: (8)                 )
1131: (4)             def find_chunks(self, tokens, **kwargs):
1132: (8)                 """Annotates the given list of tokens with chunk tags.
1133: (8)                 Several tags can be added, for example chunk + preposition tags.
1134: (8)                 """
1135: (8)                 return find_prepositions(
1136: (12)                    find_chunks(tokens, language=kwargs.get("language",
self.language))
1137: (8)                 )
1138: (4)             def find_prepositions(self, tokens, **kwargs):
1139: (8)                 """Annotates the given list of tokens with prepositional noun phrase
tags."""
1140: (8)                 return find_prepositions(tokens)  # See also Parser.find_chunks().
1141: (4)             def find_labels(self, tokens, **kwargs):
1142: (8)                 """Annotates the given list of tokens with verb/predicate tags."""
1143: (8)                 return find_relations(tokens)
1144: (4)             def find_lemmata(self, tokens, **kwargs):
1145: (8)                 """Annotates the given list of tokens with word lemmata."""
1146: (8)                 return [token + [token[0].lower()] for token in tokens]
1147: (4)             def parse(
1148: (8)                 self,
1149: (8)                 s,
1150: (8)                 tokenize=True,
```

```
1151: (8)                   tags=True,
1152: (8)                   chunks=True,
1153: (8)                   relations=False,
1154: (8)                   lemmata=False,
1155: (8)                   encoding="utf-8",
1156: (8)                   **kwargs,
1157: (4)               ):
1158: (8)                   """Takes a string (sentences) and returns a tagged Unicode string
(TaggedString).
1159: (8)                   Sentences in the output are separated by newlines.
1160: (8)                   With tokenize=True, punctuation is split from words and sentences are
separated by \n.
1161: (8)                   With tags=True, part-of-speech tags are parsed (NN, VB, IN, ...).
1162: (8)                   With chunks=True, phrase chunk tags are parsed (NP, VP, PP, PNP, ...).
1163: (8)                   With relations=True, semantic role labels are parsed (SBJ, OBJ).
1164: (8)                   With lemmata=True, word lemmata are parsed.
1165: (8)                   Optional parameters are passed to
1166: (8)                   the tokenizer, tagger, chunker, labeler and lemmatizer.
1167: (8)                   """
1168: (8)                   if tokenize:
1169: (12)                      s = self.find_tokens(s, **kwargs)
1170: (8)                   if isinstance(s, (list, tuple)):
1171: (12)                      s = [isinstance(s, basestring) and s.split(" ") or s for s in s]
1172: (8)                   if isinstance(s, basestring):
1173: (12)                      s = [s.split(" ") for s in s.split("\n")]
1174: (8)                   for i in range(len(s)):
1175: (12)                      for j in range(len(s[i])):
1176: (16)                          if isinstance(s[i][j], bytes):
1177: (20)                              s[i][j] = decode_string(s[i][j], encoding)
1178: (12)                      if tags or chunks or relations or lemmata:
1179: (16)                          s[i] = self.find_tags(s[i], **kwargs)
1180: (12)                      else:
1181: (16)                          s[i] = [[w] for w in s[i]]
1182: (12)                      if chunks or relations:
1183: (16)                          s[i] = self.find_chunks(s[i], **kwargs)
1184: (12)                      if relations:
1185: (16)                          s[i] = self.find_labels(s[i], **kwargs)
1186: (12)                      if lemmata:
1187: (16)                          s[i] = self.find_lemmata(s[i], **kwargs)
1188: (8)                   if not kwargs.get("collapse", True) or kwargs.get("split", False):
1189: (12)                      return s
1190: (8)                   format = ["word"]
1191: (8)                   if tags:
1192: (12)                      format.append("part-of-speech")
1193: (8)                   if chunks:
1194: (12)                      format.extend(("chunk", "preposition"))
1195: (8)                   if relations:
1196: (12)                      format.append("relation")
1197: (8)                   if lemmata:
1198: (12)                      format.append("lemma")
1199: (8)                   for i in range(len(s)):
1200: (12)                      for j in range(len(s[i])):
1201: (16)                          s[i][j][0] = s[i][j][0].replace("/", "&slash;")
1202: (16)                          s[i][j] = "/".join(s[i][j])
1203: (12)                      s[i] = " ".join(s[i])
1204: (8)                   s = "\n".join(s)
1205: (8)                   s = TaggedString(
1206: (12)                      str(s), format, language=kwargs.get("language", self.language)
1207: (8)                   )
1208: (8)                   return s
1209: (0)           TOKENS = "tokens"
1210: (0)           class TaggedString(str):
1211: (4)               def __new__(self, string, tags=None, language=None):
1212: (8)                   """Unicode string with tags and language attributes.
1213: (8)                   For example: TaggedString("cat/NN/NP", tags=["word", "pos", "chunk"]).
1214: (8)                   """
1215: (8)                   if tags is None:
1216: (12)                      tags = ["word"]
1217: (8)                   if isinstance(string, str) and hasattr(string, "tags"):
```

```
1218: (12)                     tags, language = string.tags, string.language
1219: (8)                 if isinstance(string, list):
1220: (12)                     string = [
1221: (16)                         [[x.replace("/", "&slash;") for x in token] for token in s]
1222: (16)                         for s in string
1223: (12)                     ]
1224: (12)                     string = "\n".join(" ".join("/".join(token) for token in s) for s
in string)
1225: (8)                 s = str.__new__(self, string)
1226: (8)                 s.tags = list(tags)
1227: (8)                 s.language = language
1228: (8)                 return s
1229: (4)             def split(self, sep=TOKENS):
1230: (8)                 """Returns a list of sentences, where each sentence is a list of
tokens,
1231: (8)                 where each token is a list of word + tags.
1232: (8)                 """
1233: (8)                 if sep != TOKENS:
1234: (12)                     return str.split(self, sep)
1235: (8)                 if len(self) == 0:
1236: (12)                     return []
1237: (8)                 return [
1238: (12)                     [
1239: (16)                         [x.replace("&slash;", "/") for x in token.split("/")]
1240: (16)                         for token in sentence.split(" ")
1241: (12)                     ]
1242: (12)                     for sentence in str.split(self, "\n")
1243: (8)                 ]
1244: (0)         class Spelling(lazydict):
1245: (4)             ALPHA = "abcdefghijklmnopqrstuvwxyz"
1246: (4)             def __init__(self, path=""):
1247: (8)                 self._path = path
1248: (4)             def load(self):
1249: (8)                 for x in _read(self._path):
1250: (12)                     x = x.split()
1251: (12)                     dict.__setitem__(self, x[0], int(x[1]))
1252: (4)             @property
1253: (4)             def path(self):
1254: (8)                 return self._path
1255: (4)             @property
1256: (4)             def language(self):
1257: (8)                 return self._language
1258: (4)             @classmethod
1259: (4)             def train(self, s, path="spelling.txt"):
1260: (8)                 """Counts the words in the given string and saves the probabilities at
the given path.
1261: (8)                 This can be used to generate a new model for the Spelling()
constructor.
1262: (8)                 """
1263: (8)                 model = {}
1264: (8)                 for w in re.findall("[a-z]+", s.lower()):
1265: (12)                     model[w] = w in model and model[w] + 1 or 1
1266: (8)                 model = (f"{k} {v}" for k, v in sorted(model.items()))
1267: (8)                 model = "\n".join(model)
1268: (8)                 f = open(path, "w")
1269: (8)                 f.write(model)
1270: (8)                 f.close()
1271: (4)             def _edit1(self, w):
1272: (8)                 """Returns a set of words with edit distance 1 from the given word."""
1273: (8)                 split = [(w[:i], w[i:]) for i in range(len(w) + 1)]
1274: (8)                 delete, transpose, replace, insert = (
1275: (12)                     [a + b[1:] for a, b in split if b],
1276: (12)                     [a + b[1] + b[0] + b[2:] for a, b in split if len(b) > 1],
1277: (12)                     [a + c + b[1:] for a, b in split for c in Spelling.ALPHA if b],
1278: (12)                     [a + c + b[0:] for a, b in split for c in Spelling.ALPHA],
1279: (8)                 )
1280: (8)                 return set(delete + transpose + replace + insert)
1281: (4)             def _edit2(self, w):
1282: (8)                 """Returns a set of words with edit distance 2 from the given word"""
```

```
1283: (8)                   return set(e2 for e1 in self._edit1(w) for e2 in self._edit1(e1) if e2
in self)
1284: (4)               def _known(self, words=None):
1285: (8)                   """Returns the given list of words filtered by known words."""
1286: (8)                   if words is None:
1287: (12)                      words = []
1288: (8)                   return set(w for w in words if w in self)
1289: (4)               def suggest(self, w):
1290: (8)                   """Return a list of (word, confidence) spelling corrections for the
given word,
1291: (8)                   based on the probability of known words with edit distance 1-2 from
the given word.
1292: (8)                   """
1293: (8)                   if len(self) == 0:
1294: (12)                      self.load()
1295: (8)                   if len(w) == 1:
1296: (12)                      return [(w, 1.0)]  # I
1297: (8)                   if w in PUNCTUATION:
1298: (12)                      return [(w, 1.0)]  # .?!
1299: (8)                   if w in string.whitespace:
1300: (12)                      return [(w, 1.0)]  # \n
1301: (8)                   if w.replace(".", "").isdigit():
1302: (12)                      return [(w, 1.0)]  # 1.5
1303: (8)                   candidates = (
1304: (12)                      self._known([w])
1305: (12)                      or self._known(self._edit1(w))
1306: (12)                      or self._known(self._edit2(w))
1307: (12)                      or [w]
1308: (8)                   )
1309: (8)                   candidates = [(self.get(c, 0.0), c) for c in candidates]
1310: (8)                   s = float(sum(p for p, word in candidates) or 1)
1311: (8)                   candidates = sorted(((p / s, word) for p, word in candidates),
reverse=True)
1312: (8)                   if w.istitle():  # Preserve capitalization
1313: (12)                      candidates = [(word.title(), p) for p, word in candidates]
1314: (8)                   else:
1315: (12)                      candidates = [(word, p) for p, word in candidates]
1316: (8)                   return candidates
```

File 5 - mixins.py:

```
1: (0)              import sys
2: (0)              class ComparableMixin:
3: (4)                  """Implements rich operators for an object."""
4: (4)                  def _compare(self, other, method):
5: (8)                      try:
6: (12)                         return method(self._cmpkey(), other._cmpkey())
7: (8)                      except (AttributeError, TypeError):
8: (12)                         return NotImplemented
9: (4)                  def __lt__(self, other):
10: (8)                     return self._compare(other, lambda s, o: s < o)
11: (4)                 def __le__(self, other):
12: (8)                     return self._compare(other, lambda s, o: s <= o)
13: (4)                 def __eq__(self, other):
14: (8)                     return self._compare(other, lambda s, o: s == o)
15: (4)                 def __ge__(self, other):
16: (8)                     return self._compare(other, lambda s, o: s >= o)
17: (4)                 def __gt__(self, other):
18: (8)                     return self._compare(other, lambda s, o: s > o)
19: (4)                 def __ne__(self, other):
20: (8)                     return self._compare(other, lambda s, o: s != o)
21: (0)             class BlobComparableMixin(ComparableMixin):
22: (4)                 """Allow blob objects to be comparable with both strings and blobs."""
23: (4)                 def _compare(self, other, method):
24: (8)                     if isinstance(other, (str, bytes)):
25: (12)                        return method(self._cmpkey(), other)
26: (8)                     return super()._compare(other, method)
```

```
(0)  class StringlikeMixin:
(4)      """Make blob objects behave like Python strings.
(4)      Expects that classes that use this mixin to have a _strkey() method that
(4)      returns the string to apply string methods to. Using _strkey() instead
(4)      of __str__ ensures consistent behavior between Python 2 and 3.
(4)      """
(4)      def __repr__(self):
(8)          """Returns a string representation for debugging."""
(8)          class_name = self.__class__.__name__
(8)          text = str(self)
(8)          return f'{class_name}("{text}")'
(4)      def __str__(self):
(8)          """Returns a string representation used in print statements
(8)          or str(my_blob)."""
(8)          return self._strkey()
(4)      def __len__(self):
(8)          """Returns the length of the raw text."""
(8)          return len(self._strkey())
(4)      def __iter__(self):
(8)          """Makes the object iterable as if it were a string,
(8)          iterating through the raw string's characters.
(8)          """
(8)          return iter(self._strkey())
(4)      def __contains__(self, sub):
(8)          """Implements the `in` keyword like a Python string."""
(8)          return sub in self._strkey()
(4)      def __getitem__(self, index):
(8)          """Returns a  substring. If index is an integer, returns a Python
(8)          string of a single character. If a range is given, e.g. `blob[3:5]`,
(8)          a new instance of the class is returned.
(8)          """
(8)          if isinstance(index, int):
(12)             return self._strkey()[index]  # Just return a single character
(8)          else:
(12)             return self.__class__(self._strkey()[index])
(4)      def find(self, sub, start=0, end=sys.maxsize):
(8)          """Behaves like the built-in str.find() method. Returns an integer,
(8)          the index of the first occurrence of the substring argument sub in the
(8)          sub-string given by [start:end].
(8)          """
(8)          return self._strkey().find(sub, start, end)
(4)      def rfind(self, sub, start=0, end=sys.maxsize):
(8)          """Behaves like the built-in str.rfind() method. Returns an integer,
(8)          the index of he last (right-most) occurence of the substring argument
(8)          sub in the sub-sequence given by [start:end].
(8)          """
(8)          return self._strkey().rfind(sub, start, end)
(4)      def index(self, sub, start=0, end=sys.maxsize):
(8)          """Like blob.find() but raise ValueError when the substring
(8)          is not found.
(8)          """
(8)          return self._strkey().index(sub, start, end)
(4)      def rindex(self, sub, start=0, end=sys.maxsize):
(8)          """Like blob.rfind() but raise ValueError when substring is not
(8)          found.
(8)          """
(8)          return self._strkey().rindex(sub, start, end)
(4)      def startswith(self, prefix, start=0, end=sys.maxsize):
(8)          """Returns True if the blob starts with the given prefix."""
(8)          return self._strkey().startswith(prefix, start, end)
(4)      def endswith(self, suffix, start=0, end=sys.maxsize):
(8)          """Returns True if the blob ends with the given suffix."""
(8)          return self._strkey().endswith(suffix, start, end)
(4)      starts_with = startswith
(4)      ends_with = endswith
(4)      def title(self):
(8)          """Returns a blob object with the text in title-case."""
(8)          return self.__class__(self._strkey().title())
(4)      def format(self, *args, **kwargs):
```

```
96: (8)                     """Perform a string formatting operation, like the built-in
97: (8)                     `str.format(*args, **kwargs)`. Returns a blob object.
98: (8)                     """
99: (8)                     return self.__class__(self._strkey().format(*args, **kwargs))
100: (4)                 def split(self, sep=None, maxsplit=sys.maxsize):
101: (8)                     """Behaves like the built-in str.split()."""
102: (8)                     return self._strkey().split(sep, maxsplit)
103: (4)                 def strip(self, chars=None):
104: (8)                     """Behaves like the built-in str.strip([chars]) method. Returns
105: (8)                     an object with leading and trailing whitespace removed.
106: (8)                     """
107: (8)                     return self.__class__(self._strkey().strip(chars))
108: (4)                 def upper(self):
109: (8)                     """Like str.upper(), returns new object with all upper-cased
characters."""
110: (8)                     return self.__class__(self._strkey().upper())
111: (4)                 def lower(self):
112: (8)                     """Like str.lower(), returns new object with all lower-cased
characters."""
113: (8)                     return self.__class__(self._strkey().lower())
114: (4)                 def join(self, iterable):
115: (8)                     """Behaves like the built-in `str.join(iterable)` method, except
116: (8)                     returns a blob object.
117: (8)                     Returns a blob which is the concatenation of the strings or blobs
118: (8)                     in the iterable.
119: (8)                     """
120: (8)                     return self.__class__(self._strkey().join(iterable))
121: (4)                 def replace(self, old, new, count=sys.maxsize):
122: (8)                     """Return a new blob object with all the occurence of `old` replaced
123: (8)                     by `new`.
124: (8)                     """
125: (8)                     return self.__class__(self._strkey().replace(old, new, count))

----------------------------------------

File 6 - formats.py:

1: (0)               """File formats for training and testing data.
2: (0)               Includes a registry of valid file formats. New file formats can be added to
the
3: (0)               registry like so: ::
4: (4)                   from textblob import formats
5: (4)                   class PipeDelimitedFormat(formats.DelimitedFormat):
6: (8)                       delimiter = '|'
7: (4)                   formats.register('psv', PipeDelimitedFormat)
8: (0)               Once a format has been registered, classifiers will be able to read data files
with
9: (0)               that format. ::
10: (4)                  from textblob.classifiers import NaiveBayesAnalyzer
11: (4)                  with open('training_data.psv', 'r') as fp:
12: (8)                      cl = NaiveBayesAnalyzer(fp, format='psv')
13: (0)              """
14: (0)              import csv
15: (0)              import json
16: (0)              from collections import OrderedDict
17: (0)              from textblob.utils import is_filelike
18: (0)              DEFAULT_ENCODING = "utf-8"
19: (0)              class BaseFormat:
20: (4)                  """Interface for format classes. Individual formats can decide on the
21: (4)                  composition and meaning of ``**kwargs``.
22: (4)                  :param File fp: A file-like object.
23: (4)                  .. versionchanged:: 0.9.0
24: (8)                      Constructor receives a file pointer rather than a file path.
25: (4)                  """
26: (4)                  def __init__(self, fp, **kwargs):
27: (8)                      pass
28: (4)                  def to_iterable(self):
29: (8)                      """Return an iterable object from the data."""
30: (8)                      raise NotImplementedError('Must implement a "to_iterable" method.')
```

```
31: (4)                 @classmethod
32: (4)                 def detect(cls, stream):
33: (8)                     """Detect the file format given a filename.
34: (8)                     Return True if a stream is this file format.
35: (8)                     .. versionchanged:: 0.9.0
36: (12)                        Changed from a static method to a class method.
37: (8)                     """
38: (8)                     raise NotImplementedError('Must implement a "detect" class method.')
39: (0)             class DelimitedFormat(BaseFormat):
40: (4)                 """A general character-delimited format."""
41: (4)                 delimiter = ","
42: (4)                 def __init__(self, fp, **kwargs):
43: (8)                     BaseFormat.__init__(self, fp, **kwargs)
44: (8)                     reader = csv.reader(fp, delimiter=self.delimiter)
45: (8)                     self.data = [row for row in reader]
46: (4)                 def to_iterable(self):
47: (8)                     """Return an iterable object from the data."""
48: (8)                     return self.data
49: (4)                 @classmethod
50: (4)                 def detect(cls, stream):
51: (8)                     """Return True if stream is valid."""
52: (8)                     try:
53: (12)                        csv.Sniffer().sniff(stream, delimiters=cls.delimiter)
54: (12)                        return True
55: (8)                     except (csv.Error, TypeError):
56: (12)                        return False
57: (0)             class CSV(DelimitedFormat):
58: (4)                 """CSV format. Assumes each row is of the form ``text,label``.
59: (4)                 ::
60: (8)                     Today is a good day,pos
61: (8)                     I hate this car.,pos
62: (4)                 """
63: (4)                 delimiter = ","
64: (0)             class TSV(DelimitedFormat):
65: (4)                 """TSV format. Assumes each row is of the form ``text\tlabel``."""
66: (4)                 delimiter = "\t"
67: (0)             class JSON(BaseFormat):
68: (4)                 """JSON format.
69: (4)                 Assumes that JSON is formatted as an array of objects with ``text`` and
70: (4)                 ``label`` properties.
71: (4)                 ::
72: (8)                     [
73: (12)                        {"text": "Today is a good day.", "label": "pos"},
74: (12)                        {"text": "I hate this car.", "label": "neg"}
75: (8)                     ]
76: (4)                 """
77: (4)                 def __init__(self, fp, **kwargs):
78: (8)                     BaseFormat.__init__(self, fp, **kwargs)
79: (8)                     self.dict = json.load(fp)
80: (4)                 def to_iterable(self):
81: (8)                     """Return an iterable object from the JSON data."""
82: (8)                     return [(d["text"], d["label"]) for d in self.dict]
83: (4)                 @classmethod
84: (4)                 def detect(cls, stream):
85: (8)                     """Return True if stream is valid JSON."""
86: (8)                     try:
87: (12)                        json.loads(stream)
88: (12)                        return True
89: (8)                     except ValueError:
90: (12)                        return False
91: (0)             _registry = OrderedDict(
92: (4)                 [
93: (8)                     ("csv", CSV),
94: (8)                     ("json", JSON),
95: (8)                     ("tsv", TSV),
96: (4)                 ]
97: (0)             )
98: (0)             def detect(fp, max_read=1024):
99: (4)                 """Attempt to detect a file's format, trying each of the supported
```

```
100: (4)                formats. Return the format class that was detected. If no format is
101: (4)                detected, return ``None``.
102: (4)                """
103: (4)                if not is_filelike(fp):
104: (8)                    return None
105: (4)                for Format in _registry.values():
106: (8)                    if Format.detect(fp.read(max_read)):
107: (12)                       fp.seek(0)
108: (12)                       return Format
109: (8)                    fp.seek(0)
110: (4)                return None
111: (0)            def get_registry():
112: (4)                """Return a dictionary of registered formats."""
113: (4)                return _registry
114: (0)            def register(name, format_class):
115: (4)                """Register a new format.
116: (4)                :param str name: The name that will be used to refer to the format, e.g.
'csv'
117: (4)                :param type format_class: The format class to register.
118: (4)                """
119: (4)                get_registry()[name] = format_class
```

File 7 - inflect.py:

```
1: (0)              """Make word inflection default to English. This allows for backwards
2: (0)              compatibility so you can still import text.inflect.
3: (4)                  >>> from textblob.inflect import singularize
4: (0)              is equivalent to
5: (4)                  >>> from textblob.en.inflect import singularize
6: (0)              """
7: (0)              from textblob.en.inflect import pluralize, singularize
8: (0)              __all__ = [
9: (4)                  "singularize",
10: (4)                 "pluralize",
11: (0)             ]
```

File 8 - parsers.py:

```
1: (0)              """Default parsers to English for backwards compatibility so you can still do
2: (0)              >>> from textblob.parsers import PatternParser
3: (0)              which is equivalent to
4: (0)              >>> from textblob.en.parsers import PatternParser
5: (0)              """
6: (0)              from textblob.base import BaseParser
7: (0)              from textblob.en.parsers import PatternParser
8: (0)              __all__ = [
9: (4)                  "BaseParser",
10: (4)                 "PatternParser",
11: (0)             ]
```

File 9 - decorators.py:

```
1: (0)              """Custom decorators."""
2: (0)              from functools import wraps
3: (0)              from textblob.exceptions import MissingCorpusError
4: (0)              class cached_property:
5: (4)                  """A property that is only computed once per instance and then replaces
6: (4)                  itself with an ordinary attribute. Deleting the attribute resets the
7: (4)                  property.
8: (4)                  Credit to Marcel Hellkamp, author of bottle.py.
9: (4)                  """
10: (4)                 def __init__(self, func):
11: (8)                     self.__doc__ = func.__doc__
```

```
12: (8)                     self.func = func
13: (4)                 def __get__(self, obj, cls):
14: (8)                     if obj is None:
15: (12)                        return self
16: (8)                     value = obj.__dict__[self.func.__name__] = self.func(obj)
17: (8)                     return value
18: (0)             def requires_nltk_corpus(func):
19: (4)                 """Wraps a function that requires an NLTK corpus. If the corpus isn't
found,
20: (4)                 raise a :exc:`MissingCorpusError`.
21: (4)                 """
22: (4)                 @wraps(func)
23: (4)                 def decorated(*args, **kwargs):
24: (8)                     try:
25: (12)                        return func(*args, **kwargs)
26: (8)                     except LookupError as error:
27: (12)                        raise MissingCorpusError() from error
28: (4)                 return decorated
```

File 10 - exceptions.py:

```
1: (0)              MISSING_CORPUS_MESSAGE = """
2: (0)              Looks like you are missing some required data for this feature.
3: (0)              To download the necessary data, simply run
4: (4)                  python -m textblob.download_corpora
5: (0)              or use the NLTK downloader to download the missing data:
http://nltk.org/data.html
6: (0)              If this doesn't fix the problem, file an issue at
https://github.com/sloria/TextBlob/issues.
7: (0)              """
8: (0)              class TextBlobError(Exception):
9: (4)                  """A TextBlob-related error."""
10: (4)                 pass
11: (0)             TextBlobException = TextBlobError  # Backwards compat
12: (0)             class MissingCorpusError(TextBlobError):
13: (4)                 """Exception thrown when a user tries to use a feature that requires a
14: (4)                 dataset or model that the user does not have on their system.
15: (4)                 """
16: (4)                 def __init__(self, message=MISSING_CORPUS_MESSAGE, *args, **kwargs):
17: (8)                     super().__init__(message, *args, **kwargs)
18: (0)             MissingCorpusException = MissingCorpusError  # Backwards compat
19: (0)             class DeprecationError(TextBlobError):
20: (4)                 """Raised when user uses a deprecated feature."""
21: (4)                 pass
22: (0)             class TranslatorError(TextBlobError):
23: (4)                 """Raised when an error occurs during language translation or
detection."""
24: (4)                 pass
25: (0)             class NotTranslated(TranslatorError):
26: (4)                 """Raised when text is unchanged after translation. This may be due to the
language
27: (4)                 being unsupported by the translator.
28: (4)                 """
29: (4)                 pass
30: (0)             class FormatError(TextBlobError):
31: (4)                 """Raised if a data file with an unsupported format is passed to a
classifier."""
32: (4)                 pass
```

File 11 - classifiers.py:

```
1: (0)              """Various classifier implementations. Also includes basic feature extractor
2: (0)              methods.
3: (0)              Example Usage:
4: (0)              ::
```

```
    >>> from textblob import TextBlob
    >>> from textblob.classifiers import NaiveBayesClassifier
    >>> train = [
    ...     ('I love this sandwich.', 'pos'),
    ...     ('This is an amazing place!', 'pos'),
    ...     ('I feel very good about these beers.', 'pos'),
    ...     ('I do not like this restaurant', 'neg'),
    ...     ('I am tired of this stuff.', 'neg'),
    ...     ("I can't deal with this", 'neg'),
    ...     ("My boss is horrible.", "neg")
    ... ]
    >>> cl = NaiveBayesClassifier(train)
    >>> cl.classify("I feel amazing!")
    'pos'
    >>> blob = TextBlob("The beer is good. But the hangover is horrible.", classifier=cl)
    >>> for s in blob.sentences:
    ...     print(s)
    ...     print(s.classify())
    ...
    The beer is good.
    pos
    But the hangover is horrible.
    neg
.. versionadded:: 0.6.0
"""  # noqa: E501
from itertools import chain
import nltk
import textblob.formats as formats
from textblob.decorators import cached_property
from textblob.exceptions import FormatError
from textblob.tokenizers import word_tokenize
from textblob.utils import is_filelike, strip_punc
basestring = (str, bytes)
def _get_words_from_dataset(dataset):
    """Return a set of all words in a dataset.
    :param dataset: A list of tuples of the form ``(words, label)`` where
        ``words`` is either a string of a list of tokens.
    """
    def tokenize(words):
        if isinstance(words, basestring):
            return word_tokenize(words, include_punc=False)
        else:
            return words
    all_words = chain.from_iterable(tokenize(words) for words, _ in dataset)
    return set(all_words)
def _get_document_tokens(document):
    if isinstance(document, basestring):
        tokens = set(
            strip_punc(w, all=False)
            for w in word_tokenize(document, include_punc=False)
        )
    else:
        tokens = set(strip_punc(w, all=False) for w in document)
    return tokens
def basic_extractor(document, train_set):
    """A basic document feature extractor that returns a dict indicating
    what words in ``train_set`` are contained in ``document``.
    :param document: The text to extract features from. Can be a string or an iterable.
    :param list train_set: Training data set, a list of tuples of the form
        ``(words, label)`` OR an iterable of strings.
    """
    try:
        el_zero = next(iter(train_set))  # Infer input from first element.
    except StopIteration:
        return {}
    if isinstance(el_zero, basestring):
        word_features = [w for w in chain([el_zero], train_set)]
```

```
72: (4)                 else:
73: (8)                     try:
74: (12)                        assert isinstance(el_zero[0], basestring)
75: (12)                        word_features = _get_words_from_dataset(chain([el_zero],
train_set))
76: (8)                     except Exception as error:
77: (12)                        raise ValueError("train_set is probably malformed.") from error
78: (4)                 tokens = _get_document_tokens(document)
79: (4)                 features = dict((f"contains({word})", (word in tokens)) for word in
word_features)
80: (4)                 return features
81: (0)             def contains_extractor(document):
82: (4)                 """A basic document feature extractor that returns a dict of words that
83: (4)                 the document contains.
84: (4)                 """
85: (4)                 tokens = _get_document_tokens(document)
86: (4)                 features = dict((f"contains({w})", True) for w in tokens)
87: (4)                 return features
88: (0)             class BaseClassifier:
89: (4)                 """Abstract classifier class from which all classifers inherit. At a
90: (4)                 minimum, descendant classes must implement a ``classify`` method and have
91: (4)                 a ``classifier`` property.
92: (4)                 :param train_set: The training set, either a list of tuples of the form
93: (8)                     ``(text, classification)`` or a file-like object. ``text`` may be
either
94: (8)                     a string or an iterable.
95: (4)                 :param callable feature_extractor: A feature extractor function that takes
one or
96: (8)                     two arguments: ``document`` and ``train_set``.
97: (4)                 :param str format: If ``train_set`` is a filename, the file format, e.g.
98: (8)                     ``"csv"`` or ``"json"``. If ``None``, will attempt to detect the
99: (8)                     file format.
100: (4)                :param kwargs: Additional keyword arguments are passed to the constructor
101: (8)                    of the :class:`Format <textblob.formats.BaseFormat>` class used to
102: (8)                    read the data. Only applies when a file-like object is passed as
103: (8)                    ``train_set``.
104: (4)                .. versionadded:: 0.6.0
105: (4)                """
106: (4)                def __init__(
107: (8)                    self, train_set, feature_extractor=basic_extractor, format=None,
**kwargs
108: (4)                ):
109: (8)                    self.format_kwargs = kwargs
110: (8)                    self.feature_extractor = feature_extractor
111: (8)                    if is_filelike(train_set):
112: (12)                       self.train_set = self._read_data(train_set, format)
113: (8)                    else:  # train_set is a list of tuples
114: (12)                       self.train_set = train_set
115: (8)                    self._word_set = _get_words_from_dataset(
116: (12)                       self.train_set
117: (8)                    )  # Keep a hidden set of unique words.
118: (8)                    self.train_features = None
119: (4)                def _read_data(self, dataset, format=None):
120: (8)                    """Reads a data file and returns an iterable that can be used
121: (8)                    as testing or training data.
122: (8)                    """
123: (8)                    if not format:
124: (12)                       format_class = formats.detect(dataset)
125: (12)                       if not format_class:
126: (16)                           raise FormatError(
127: (20)                               "Could not automatically detect format for the given "
128: (20)                               "data source."
129: (16)                           )
130: (8)                    else:
131: (12)                       registry = formats.get_registry()
132: (12)                       if format not in registry.keys():
133: (16)                           raise ValueError(f"'{format}' format not supported.")
134: (12)                       format_class = registry[format]
135: (8)                    return format_class(dataset, **self.format_kwargs).to_iterable()
```

```
136: (4)                @cached_property
137: (4)                def classifier(self):
138: (8)                    """The classifier object."""
139: (8)                    raise NotImplementedError('Must implement the "classifier" property.')
140: (4)                def classify(self, text):
141: (8)                    """Classifies a string of text."""
142: (8)                    raise NotImplementedError('Must implement a "classify" method.')
143: (4)                def train(self, labeled_featureset):
144: (8)                    """Trains the classifier."""
145: (8)                    raise NotImplementedError('Must implement a "train" method.')
146: (4)                def labels(self):
147: (8)                    """Returns an iterable containing the possible labels."""
148: (8)                    raise NotImplementedError('Must implement a "labels" method.')
149: (4)                def extract_features(self, text):
150: (8)                    """Extracts features from a body of text.
151: (8)                    :rtype: dictionary of features
152: (8)                    """
153: (8)                    try:
154: (12)                       return self.feature_extractor(text, self._word_set)
155: (8)                    except (TypeError, AttributeError):
156: (12)                       return self.feature_extractor(text)
157: (0)            class NLTKClassifier(BaseClassifier):
158: (4)                """An abstract class that wraps around the nltk.classify module.
159: (4)                Expects that descendant classes include a class variable ``nltk_class``
160: (4)                which is the class in the nltk.classify module to be wrapped.
161: (4)                Example: ::
162: (8)                    class MyClassifier(NLTKClassifier):
163: (12)                       nltk_class = nltk.classify.svm.SvmClassifier
164: (4)                """
165: (4)                nltk_class = None
166: (4)                def __init__(
167: (8)                    self, train_set, feature_extractor=basic_extractor, format=None,
     **kwargs
168: (4)                ):
169: (8)                    super().__init__(train_set, feature_extractor, format, **kwargs)
170: (8)                    self.train_features = [(self.extract_features(d), c) for d, c in
     self.train_set]
171: (4)                def __repr__(self):
172: (8)                    class_name = self.__class__.__name__
173: (8)                    return f"<{class_name} trained on {len(self.train_set)} instances>"
174: (4)                @cached_property
175: (4)                def classifier(self):
176: (8)                    """The classifier."""
177: (8)                    try:
178: (12)                       return self.train()
179: (8)                    except AttributeError as error:  # nltk_class has not been defined
180: (12)                       raise ValueError(
181: (16)                           "NLTKClassifier must have a nltk_class" " variable that is not
     None."
182: (12)                       ) from error
183: (4)                def train(self, *args, **kwargs):
184: (8)                    """Train the classifier with a labeled feature set and return
185: (8)                    the classifier. Takes the same arguments as the wrapped NLTK class.
186: (8)                    This method is implicitly called when calling ``classify`` or
187: (8)                    ``accuracy`` methods and is included only to allow passing in
     arguments
188: (8)                    to the ``train`` method of the wrapped NLTK class.
189: (8)                    .. versionadded:: 0.6.2
190: (8)                    :rtype: A classifier
191: (8)                    """
192: (8)                    try:
193: (12)                       self.classifier = self.nltk_class.train(
194: (16)                           self.train_features, *args, **kwargs
195: (12)                       )
196: (12)                       return self.classifier
197: (8)                    except AttributeError as error:
198: (12)                       raise ValueError(
199: (16)                           "NLTKClassifier must have a nltk_class" " variable that is not
     None."
```

```
200: (12)                     ) from error
201: (4)                 def labels(self):
202: (8)                     """Return an iterable of possible labels."""
203: (8)                     return self.classifier.labels()
204: (4)                 def classify(self, text):
205: (8)                     """Classifies the text.
206: (8)                     :param str text: A string of text.
207: (8)                     """
208: (8)                     text_features = self.extract_features(text)
209: (8)                     return self.classifier.classify(text_features)
210: (4)                 def accuracy(self, test_set, format=None):
211: (8)                     """Compute the accuracy on a test set.
212: (8)                     :param test_set: A list of tuples of the form ``(text, label)``, or a
213: (12)                        file pointer.
214: (8)                     :param format: If ``test_set`` is a filename, the file format, e.g.
215: (12)                        ``"csv"`` or ``"json"``. If ``None``, will attempt to detect the
216: (12)                        file format.
217: (8)                     """
218: (8)                     if is_filelike(test_set):
219: (12)                        test_data = self._read_data(test_set, format)
220: (8)                     else:  # test_set is a list of tuples
221: (12)                        test_data = test_set
222: (8)                     test_features = [(self.extract_features(d), c) for d, c in test_data]
223: (8)                     return nltk.classify.accuracy(self.classifier, test_features)
224: (4)                 def update(self, new_data, *args, **kwargs):
225: (8)                     """Update the classifier with new training data and re-trains the
226: (8)                     classifier.
227: (8)                     :param new_data: New data as a list of tuples of the form
228: (12)                        ``(text, label)``.
229: (8)                     """
230: (8)                     self.train_set += new_data
231: (8)                     self._word_set.update(_get_words_from_dataset(new_data))
232: (8)                     self.train_features = [(self.extract_features(d), c) for d, c in
self.train_set]
233: (8)                     try:
234: (12)                        self.classifier = self.nltk_class.train(
235: (16)                            self.train_features, *args, **kwargs
236: (12)                        )
237: (8)                     except AttributeError as error:  # Descendant has not defined
nltk_class
238: (12)                        raise ValueError(
239: (16)                            "NLTKClassifier must have a nltk_class" " variable that is not
None."
240: (12)                        ) from error
241: (8)                     return True
242: (0)             class NaiveBayesClassifier(NLTKClassifier):
243: (4)                 """A classifier based on the Naive Bayes algorithm, as implemented in
244: (4)                 NLTK.
245: (4)                 :param train_set: The training set, either a list of tuples of the form
246: (8)                     ``(text, classification)`` or a filename. ``text`` may be either
247: (8)                     a string or an iterable.
248: (4)                 :param feature_extractor: A feature extractor function that takes one or
249: (8)                     two arguments: ``document`` and ``train_set``.
250: (4)                 :param format: If ``train_set`` is a filename, the file format, e.g.
251: (8)                     ``"csv"`` or ``"json"``. If ``None``, will attempt to detect the
252: (8)                     file format.
253: (4)                 .. versionadded:: 0.6.0
254: (4)                 """
255: (4)                 nltk_class = nltk.classify.NaiveBayesClassifier
256: (4)                 def prob_classify(self, text):
257: (8)                     """Return the label probability distribution for classifying a string
258: (8)                     of text.
259: (8)                     Example:
260: (8)                     ::
261: (12)                        >>> classifier = NaiveBayesClassifier(train_data)
262: (12)                        >>> prob_dist = classifier.prob_classify("I feel happy this
morning.")
263: (12)                        >>> prob_dist.max()
264: (12)                        'positive'
```

```
265: (12)                     >>> prob_dist.prob("positive")
266: (12)                     0.7
267: (8)                  :rtype: nltk.probability.DictionaryProbDist
268: (8)                  """
269: (8)                  text_features = self.extract_features(text)
270: (8)                  return self.classifier.prob_classify(text_features)
271: (4)              def informative_features(self, *args, **kwargs):
272: (8)                  """Return the most informative features as a list of tuples of the
273: (8)                  form ``(feature_name, feature_value)``.
274: (8)                  :rtype: list
275: (8)                  """
276: (8)                  return self.classifier.most_informative_features(*args, **kwargs)
277: (4)              def show_informative_features(self, *args, **kwargs):
278: (8)                  """Displays a listing of the most informative features for this
279: (8)                  classifier.
280: (8)                  :rtype: None
281: (8)                  """
282: (8)                  return self.classifier.show_most_informative_features(*args, **kwargs)
283: (0)          class DecisionTreeClassifier(NLTKClassifier):
284: (4)              """A classifier based on the decision tree algorithm, as implemented in
285: (4)              NLTK.
286: (4)              :param train_set: The training set, either a list of tuples of the form
287: (8)                  ``(text, classification)`` or a filename. ``text`` may be either
288: (8)                  a string or an iterable.
289: (4)              :param feature_extractor: A feature extractor function that takes one or
290: (8)                  two arguments: ``document`` and ``train_set``.
291: (4)              :param format: If ``train_set`` is a filename, the file format, e.g.
292: (8)                  ``"csv"`` or ``"json"``. If ``None``, will attempt to detect the
293: (8)                  file format.
294: (4)              .. versionadded:: 0.6.2
295: (4)              """
296: (4)              nltk_class = nltk.classify.decisiontree.DecisionTreeClassifier
297: (4)              def pretty_format(self, *args, **kwargs):
298: (8)                  """Return a string containing a pretty-printed version of this
decision
299: (8)                  tree. Each line in the string corresponds to a single decision tree
node
300: (8)                  or leaf, and indentation is used to display the structure of the tree.
301: (8)                  :rtype: str
302: (8)                  """
303: (8)                  return self.classifier.pretty_format(*args, **kwargs)
304: (4)              pprint = pretty_format
305: (4)              def pseudocode(self, *args, **kwargs):
306: (8)                  """Return a string representation of this decision tree that expresses
307: (8)                  the decisions it makes as a nested set of pseudocode if statements.
308: (8)                  :rtype: str
309: (8)                  """
310: (8)                  return self.classifier.pseudocode(*args, **kwargs)
311: (0)          class PositiveNaiveBayesClassifier(NLTKClassifier):
312: (4)              """A variant of the Naive Bayes Classifier that performs binary
313: (4)              classification with partially-labeled training sets, i.e. when only
314: (4)              one class is labeled and the other is not. Assuming a prior distribution
315: (4)              on the two labels, uses the unlabeled set to estimate the frequencies of
316: (4)              the features.
317: (4)              Example usage:
318: (4)              ::
319: (8)                  >>> from text.classifiers import PositiveNaiveBayesClassifier
320: (8)                  >>> sports_sentences = ['The team dominated the game',
321: (8)                  ...                     'They lost the ball',
322: (8)                  ...                     'The game was intense',
323: (8)                  ...                     'The goalkeeper catched the ball',
324: (8)                  ...                     'The other team controlled the ball']
325: (8)                  >>> various_sentences = ['The President did not comment',
326: (8)                  ...                       'I lost the keys',
327: (8)                  ...                       'The team won the game',
328: (8)                  ...                       'Sara has two kids',
329: (8)                  ...                       'The ball went off the court',
330: (8)                  ...                       'They had the ball for the whole game',
331: (8)                  ...                       'The show is over']
```

```
                    >>> classifier =
PositiveNaiveBayesClassifier(positive_set=sports_sentences,
                    ...
unlabeled_set=various_sentences)
                    >>> classifier.classify("My team lost the game")
                    True
                    >>> classifier.classify("And now for something completely different.")
                    False
                :param positive_set: A collection of strings that have the positive label.
                :param unlabeled_set: A collection of unlabeled strings.
                :param feature_extractor: A feature extractor function.
                :param positive_prob_prior: A prior estimate of the probability of the
                    label ``True``.
                .. versionadded:: 0.7.0
                """
                nltk_class = nltk.classify.PositiveNaiveBayesClassifier
                def __init__(
                    self,
                    positive_set,
                    unlabeled_set,
                    feature_extractor=contains_extractor,
                    positive_prob_prior=0.5,
                    **kwargs,
                ):
                    self.feature_extractor = feature_extractor
                    self.positive_set = positive_set
                    self.unlabeled_set = unlabeled_set
                    self.positive_features = [self.extract_features(d) for d in
self.positive_set]
                    self.unlabeled_features = [self.extract_features(d) for d in
self.unlabeled_set]
                    self.positive_prob_prior = positive_prob_prior
                def __repr__(self):
                    class_name = self.__class__.__name__
                    return (
                        f"<{class_name} trained on {len(self.positive_set)} labeled "
                        f"and {len(self.unlabeled_set)} unlabeled instances>"
                    )
                def train(self, *args, **kwargs):
                    """Train the classifier with a labeled and unlabeled feature sets and
return
                    the classifier. Takes the same arguments as the wrapped NLTK class.
                    This method is implicitly called when calling ``classify`` or
                    ``accuracy`` methods and is included only to allow passing in
arguments
                    to the ``train`` method of the wrapped NLTK class.
                    :rtype: A classifier
                    """
                    self.classifier = self.nltk_class.train(
                        self.positive_features, self.unlabeled_features,
self.positive_prob_prior
                    )
                    return self.classifier
                def update(
                    self,
                    new_positive_data=None,
                    new_unlabeled_data=None,
                    positive_prob_prior=0.5,
                    *args,
                    **kwargs,
                ):
                    """Update the classifier with new data and re-trains the
                    classifier.
                    :param new_positive_data: List of new, labeled strings.
                    :param new_unlabeled_data: List of new, unlabeled strings.
                    """
                    self.positive_prob_prior = positive_prob_prior
                    if new_positive_data:
                        self.positive_set += new_positive_data
```

```
394: (12)                         self.positive_features += [
395: (16)                             self.extract_features(d) for d in new_positive_data
396: (12)                         ]
397: (8)                      if new_unlabeled_data:
398: (12)                         self.unlabeled_set += new_unlabeled_data
399: (12)                         self.unlabeled_features += [
400: (16)                             self.extract_features(d) for d in new_unlabeled_data
401: (12)                         ]
402: (8)                      self.classifier = self.nltk_class.train(
403: (12)                         self.positive_features,
404: (12)                         self.unlabeled_features,
405: (12)                         self.positive_prob_prior,
406: (12)                         *args,
407: (12)                         **kwargs,
408: (8)                      )
409: (8)                      return True
410: (0)              class MaxEntClassifier(NLTKClassifier):
411: (4)                  __doc__ = nltk.classify.maxent.MaxentClassifier.__doc__
412: (4)                  nltk_class = nltk.classify.maxent.MaxentClassifier
413: (4)                  def prob_classify(self, text):
414: (8)                      """Return the label probability distribution for classifying a string
415: (8)                      of text.
416: (8)                      Example:
417: (8)                      ::
418: (12)                         >>> classifier = MaxEntClassifier(train_data)
419: (12)                         >>> prob_dist = classifier.prob_classify("I feel happy this
morning.")
420: (12)                         >>> prob_dist.max()
421: (12)                         'positive'
422: (12)                         >>> prob_dist.prob("positive")
423: (12)                         0.7
424: (8)                      :rtype: nltk.probability.DictionaryProbDist
425: (8)                      """
426: (8)                      feats = self.extract_features(text)
427: (8)                      return self.classifier.prob_classify(feats)
```

File 12 - np_extractors.py:

```
1: (0)              """Default noun phrase extractors are for English to maintain backwards
2: (0)              compatibility, so you can still do
3: (0)              >>> from textblob.np_extractors import ConllExtractor
4: (0)              which is equivalent to
5: (0)              >>> from textblob.en.np_extractors import ConllExtractor
6: (0)              """
7: (0)              from textblob.base import BaseNPExtractor
8: (0)              from textblob.en.np_extractors import ConllExtractor, FastNPExtractor
9: (0)              __all__ = [
10: (4)                 "BaseNPExtractor",
11: (4)                 "ConllExtractor",
12: (4)                 "FastNPExtractor",
13: (0)             ]
```

File 13 - download_corpora.py:

```
1: (0)              """Downloads the necessary NLTK corpora for TextBlob.
2: (0)              Usage: ::
3: (4)                  $ python -m textblob.download_corpora
4: (0)              If you only intend to use TextBlob's default models, you can use the "lite"
5: (0)              option: ::
6: (4)                  $ python -m textblob.download_corpora lite
7: (0)              """
8: (0)              import sys
9: (0)              import nltk
10: (0)             MIN_CORPORA = [
11: (4)                 "brown",  # Required for FastNPExtractor
```

```
12: (4)                 "punkt",  # Required for WordTokenizer
13: (4)                 "wordnet",  # Required for lemmatization
14: (4)                 "averaged_perceptron_tagger",  # Required for NLTKTagger
15: (0)             ]
16: (0)             ADDITIONAL_CORPORA = [
17: (4)                 "conll2000",  # Required for ConllExtractor
18: (4)                 "movie_reviews",  # Required for NaiveBayesAnalyzer
19: (0)             ]
20: (0)             ALL_CORPORA = MIN_CORPORA + ADDITIONAL_CORPORA
21: (0)             def download_lite():
22: (4)                 for each in MIN_CORPORA:
23: (8)                     nltk.download(each)
24: (0)             def download_all():
25: (4)                 for each in ALL_CORPORA:
26: (8)                     nltk.download(each)
27: (0)             def main():
28: (4)                 if "lite" in sys.argv:
29: (8)                     download_lite()
30: (4)                 else:
31: (8)                     download_all()
32: (4)                 print("Finished.")
33: (0)             if __name__ == "__main__":
34: (4)                 main()
```

File 14 - utils.py:

```
1: (0)              import re
2: (0)              import string
3: (0)              PUNCTUATION_REGEX = re.compile(f"[{re.escape(string.punctuation)}]")
4: (0)              def strip_punc(s, all=False):
5: (4)                  """Removes punctuation from a string.
6: (4)                  :param s: The string.
7: (4)                  :param all: Remove all punctuation. If False, only removes punctuation
from
8: (8)                      the ends of the string.
9: (4)                  """
10: (4)                 if all:
11: (8)                     return PUNCTUATION_REGEX.sub("", s.strip())
12: (4)                 else:
13: (8)                     return s.strip().strip(string.punctuation)
14: (0)             def lowerstrip(s, all=False):
15: (4)                 """Makes text all lowercase and strips punctuation and whitespace.
16: (4)                 :param s: The string.
17: (4)                 :param all: Remove all punctuation. If False, only removes punctuation
from
18: (8)                     the ends of the string.
19: (4)                 """
20: (4)                 return strip_punc(s.lower().strip(), all=all)
21: (0)             def tree2str(tree, concat=" "):
22: (4)                 """Convert a nltk.tree.Tree to a string.
23: (4)                 For example:
24: (8)                     (NP a/DT beautiful/JJ new/JJ dashboard/NN) -> "a beautiful dashboard"
25: (4)                 """
26: (4)                 return concat.join([word for (word, tag) in tree])
27: (0)             def filter_insignificant(chunk, tag_suffixes=("DT", "CC", "PRP$", "PRP")):
28: (4)                 """Filter out insignificant (word, tag) tuples from a chunk of text."""
29: (4)                 good = []
30: (4)                 for word, tag in chunk:
31: (8)                     ok = True
32: (8)                     for suffix in tag_suffixes:
33: (12)                        if tag.endswith(suffix):
34: (16)                            ok = False
35: (16)                            break
36: (8)                     if ok:
37: (12)                        good.append((word, tag))
38: (4)                 return good
39: (0)             def is_filelike(obj):
```

```
40: (4)                 """Return whether ``obj`` is a file-like object."""
41: (4)                 return hasattr(obj, "read")
```

File 15 - taggers.py:

```
1: (0)              """Default taggers to the English taggers for backwards incompatibility, so
you
2: (0)              can still do
3: (0)              >>> from textblob.taggers import NLTKTagger
4: (0)              which is equivalent to
5: (0)              >>> from textblob.en.taggers import NLTKTagger
6: (0)              """
7: (0)              from textblob.base import BaseTagger
8: (0)              from textblob.en.taggers import NLTKTagger, PatternTagger
9: (0)              __all__ = [
10: (4)                 "BaseTagger",
11: (4)                 "PatternTagger",
12: (4)                 "NLTKTagger",
13: (0)             ]
```

File 16 - wordnet.py:

```
1: (0)              """Wordnet interface. Contains classes for creating Synsets and Lemmas
2: (0)              directly.
3: (0)              .. versionadded:: 0.7.0
4: (0)              """
5: (0)              import nltk
6: (0)              wordnet = nltk.corpus.wordnet
7: (0)              Synset = nltk.corpus.wordnet.synset
8: (0)              Lemma = nltk.corpus.wordnet.lemma
9: (0)              VERB, NOUN, ADJ, ADV = wordnet.VERB, wordnet.NOUN, wordnet.ADJ, wordnet.ADV
```

File 17 - __init__.py:

```
1: (0)              """This file is based on pattern.en. See the bundled NOTICE file for
2: (0)              license information.
3: (0)              """
4: (0)              import os
5: (0)              from textblob._text import CHUNK, PENN, PNP, POS, UNIVERSAL, WORD, Lexicon,
Spelling
6: (0)              from textblob._text import Parser as _Parser
7: (0)              from textblob._text import Sentiment as _Sentiment
8: (0)              try:
9: (4)                  MODULE = os.path.dirname(os.path.abspath(__file__))
10: (0)             except:
11: (4)                 MODULE = ""
12: (0)             spelling = Spelling(path=os.path.join(MODULE, "en-spelling.txt"))
13: (0)             def find_lemmata(tokens):
14: (4)                 """Annotates the tokens with lemmata for plural nouns and conjugated
verbs,
15: (4)                 where each token is a [word, part-of-speech] list.
16: (4)                 """
17: (4)                 for token in tokens:
18: (8)                     word, pos, lemma = token[0], token[1], token[0]
19: (8)                     if pos == "NNS":
20: (12)                        lemma = singularize(word)
21: (8)                     if pos.startswith(("VB", "MD")):
22: (12)                        lemma = conjugate(word, INFINITIVE) or word
23: (8)                     token.append(lemma.lower())
24: (4)                 return tokens
25: (0)             class Parser(_Parser):
26: (4)                 def find_lemmata(self, tokens, **kwargs):
27: (8)                     return find_lemmata(tokens)
```

```
28: (4)                 def find_tags(self, tokens, **kwargs):
29: (8)                     if kwargs.get("tagset") in (PENN, None):
30: (12)                        kwargs.setdefault("map", lambda token, tag: (token, tag))
31: (8)                     if kwargs.get("tagset") == UNIVERSAL:
32: (12)                        kwargs.setdefault(
33: (16)                            "map", lambda token, tag: penntreebank2universal(token, tag)
34: (12)                        )
35: (8)                     return _Parser.find_tags(self, tokens, **kwargs)
36: (0)             class Sentiment(_Sentiment):
37: (4)                 def load(self, path=None):
38: (8)                     _Sentiment.load(self, path)
39: (8)                     if not path:
40: (12)                        for w, pos in list(dict.items(self)):
41: (16)                            if "JJ" in pos:
42: (20)                                if w.endswith("y"):
43: (24)                                    w = w[:-1] + "i"
44: (20)                                if w.endswith("le"):
45: (24)                                    w = w[:-2]
46: (20)                                p, s, i = pos["JJ"]
47: (20)                                self.annotate(w + "ly", "RB", p, s, i)
48: (0)             lexicon = Lexicon(
49: (4)                 path=os.path.join(MODULE, "en-lexicon.txt"),
50: (4)                 morphology=os.path.join(MODULE, "en-morphology.txt"),
51: (4)                 context=os.path.join(MODULE, "en-context.txt"),
52: (4)                 entities=os.path.join(MODULE, "en-entities.txt"),
53: (4)                 language="en",
54: (0)             )
55: (0)             parser = Parser(lexicon=lexicon, default=("NN", "NNP", "CD"), language="en")
56: (0)             sentiment = Sentiment(
57: (4)                 path=os.path.join(MODULE, "en-sentiment.xml"),
58: (4)                 synset="wordnet_id",
59: (4)                 negations=("no", "not", "n't", "never"),
60: (4)                 modifiers=("RB",),
61: (4)                 modifier=lambda w: w.endswith("ly"),
62: (4)                 tokenizer=parser.find_tokens,
63: (4)                 language="en",
64: (0)             )
65: (0)             def tokenize(s, *args, **kwargs):
66: (4)                 """Returns a list of sentences, where punctuation marks have been split
from words."""
67: (4)                 return parser.find_tokens(str(s), *args, **kwargs)
68: (0)             def parse(s, *args, **kwargs):
69: (4)                 """Returns a tagged str string."""
70: (4)                 return parser.parse(str(s), *args, **kwargs)
71: (0)             def parsetree(s, *args, **kwargs):
72: (4)                 """Returns a parsed Text from the given string."""
73: (4)                 return Text(parse(str(s), *args, **kwargs))
74: (0)             def split(s, token=None):
75: (4)                 """Returns a parsed Text from the given parsed string."""
76: (4)                 if token is None:
77: (8)                     token = [WORD, POS, CHUNK, PNP]
78: (4)                 return Text(str(s), token)
79: (0)             def tag(s, tokenize=True, encoding="utf-8"):
80: (4)                 """Returns a list of (token, tag)-tuples from the given string."""
81: (4)                 tags = []
82: (4)                 for sentence in parse(s, tokenize, True, False, False, False,
encoding).split():
83: (8)                     for token in sentence:
84: (12)                        tags.append((token[0], token[1]))
85: (4)                 return tags
86: (0)             def suggest(w):
87: (4)                 """Returns a list of (word, confidence)-tuples of spelling corrections."""
88: (4)                 return spelling.suggest(w)
89: (0)             def polarity(s, **kwargs):
90: (4)                 """Returns the sentence polarity (positive/negative) between -1.0 and
1.0."""
91: (4)                 return sentiment(str(s), **kwargs)[0]
92: (0)             def subjectivity(s, **kwargs):
93: (4)                 """Returns the sentence subjectivity (objective/subjective) between 0.0
```

```
and 1.0."""
94: (4)                 return sentiment(str(s), **kwargs)[1]
95: (0)             def positive(s, threshold=0.1, **kwargs):
96: (4)                 """Returns True if the given sentence has a positive sentiment (polarity
>= threshold)."""
97: (4)                 return polarity(str(s), **kwargs) >= threshold
```

File 18 - sentiments.py:

```
1: (0)              """Default sentiment analyzers are English for backwards compatibility, so
2: (0)              you can still do
3: (0)              >>> from textblob.sentiments import PatternAnalyzer
4: (0)              which is equivalent to
5: (0)              >>> from textblob.en.sentiments import PatternAnalyzer
6: (0)              """
7: (0)              from textblob.base import BaseSentimentAnalyzer
8: (0)              from textblob.en.sentiments import (
9: (4)                  CONTINUOUS,
10: (4)                 DISCRETE,
11: (4)                 NaiveBayesAnalyzer,
12: (4)                 PatternAnalyzer,
13: (0)             )
14: (0)             __all__ = [
15: (4)                 "BaseSentimentAnalyzer",
16: (4)                 "DISCRETE",
17: (4)                 "CONTINUOUS",
18: (4)                 "PatternAnalyzer",
19: (4)                 "NaiveBayesAnalyzer",
20: (0)             ]
```

File 19 - tokenizers.py:

```
1: (0)              """Various tokenizer implementations.
2: (0)              .. versionadded:: 0.4.0
3: (0)              """
4: (0)              from itertools import chain
5: (0)              import nltk
6: (0)              from textblob.base import BaseTokenizer
7: (0)              from textblob.decorators import requires_nltk_corpus
8: (0)              from textblob.utils import strip_punc
9: (0)              class WordTokenizer(BaseTokenizer):
10: (4)                 """NLTK's recommended word tokenizer (currently the TreeBankTokenizer).
11: (4)                 Uses regular expressions to tokenize text. Assumes text has already been
12: (4)                 segmented into sentences.
13: (4)                 Performs the following steps:
14: (4)                 * split standard contractions, e.g. don't -> do n't
15: (4)                 * split commas and single quotes
16: (4)                 * separate periods that appear at the end of line
17: (4)                 """
18: (4)                 def tokenize(self, text, include_punc=True):
19: (8)                     """Return a list of word tokens.
20: (8)                     :param text: string of text.
21: (8)                     :param include_punc: (optional) whether to
22: (12)                        include punctuation as separate tokens. Default to True.
23: (8)                     """
24: (8)                     tokens = nltk.tokenize.word_tokenize(text)
25: (8)                     if include_punc:
26: (12)                        return tokens
27: (8)                     else:
28: (12)                        return [
29: (16)                            word if word.startswith("'") else strip_punc(word, all=False)
30: (16)                            for word in tokens
31: (16)                            if strip_punc(word, all=False)
32: (12)                        ]
33: (0)             class SentenceTokenizer(BaseTokenizer):
```

```
34: (4)                 """NLTK's sentence tokenizer (currently PunktSentenceTokenizer).
35: (4)                 Uses an unsupervised algorithm to build a model for abbreviation words,
36: (4)                 collocations, and words that start sentences,
37: (4)                 then uses that to find sentence boundaries.
38: (4)                 """
39: (4)                 @requires_nltk_corpus
40: (4)                 def tokenize(self, text):
41: (8)                     """Return a list of sentences."""
42: (8)                     return nltk.tokenize.sent_tokenize(text)
43: (0)             sent_tokenize = SentenceTokenizer().itokenize
44: (0)             _word_tokenizer = WordTokenizer()  # Singleton word tokenizer
45: (0)             def word_tokenize(text, include_punc=True, *args, **kwargs):
46: (4)                 """Convenience function for tokenizing text into words.
47: (4)                 NOTE: NLTK's word tokenizer expects sentences as input, so the text will
be
48: (4)                 tokenized to sentences before being tokenized to words.
49: (4)                 """
50: (4)                 words = chain.from_iterable(
51: (8)                     _word_tokenizer.itokenize(sentence, include_punc, *args, **kwargs)
52: (8)                     for sentence in sent_tokenize(text)
53: (4)                 )
54: (4)                 return words
```

File 20 - inflect.py:

```
1: (0)              """The pluralize and singular methods from the pattern library.
2: (0)              Licenced under the BSD.
3: (0)              See here https://github.com/clips/pattern/blob/master/LICENSE.txt for
4: (0)              complete license information.
5: (0)              """
6: (0)              import re
7: (0)              VERB, NOUN, ADJECTIVE, ADVERB = "VB", "NN", "JJ", "RB"
8: (0)              plural_prepositions = [
9: (4)                  "about",
10: (4)                 "above",
11: (4)                 "across",
12: (4)                 "after",
13: (4)                 "among",
14: (4)                 "around",
15: (4)                 "at",
16: (4)                 "athwart",
17: (4)                 "before",
18: (4)                 "behind",
19: (4)                 "below",
20: (4)                 "beneath",
21: (4)                 "beside",
22: (4)                 "besides",
23: (4)                 "between",
24: (4)                 "betwixt",
25: (4)                 "beyond",
26: (4)                 "but",
27: (4)                 "by",
28: (4)                 "during",
29: (4)                 "except",
30: (4)                 "for",
31: (4)                 "from",
32: (4)                 "in",
33: (4)                 "into",
34: (4)                 "near",
35: (4)                 "of",
36: (4)                 "off",
37: (4)                 "on",
38: (4)                 "onto",
39: (4)                 "out",
40: (4)                 "over",
41: (4)                 "since",
42: (4)                 "till",
```

```
 43: (4)                 "to",
 44: (4)                 "under",
 45: (4)                 "until",
 46: (4)                 "unto",
 47: (4)                 "upon",
 48: (4)                 "with",
 49: (0)             ]
 50: (0)             plural_rules = [
 51: (4)                 [
 52: (8)                     ["^a$|^an$", "some", None, False],
 53: (8)                     ["^this$", "these", None, False],
 54: (8)                     ["^that$", "those", None, False],
 55: (8)                     ["^any$", "all", None, False],
 56: (4)                 ],
 57: (4)                 [
 58: (8)                     ["^my$", "our", None, False],
 59: (8)                     ["^your$|^thy$", "your", None, False],
 60: (8)                     ["^her$|^his$|^its$|^their$", "their", None, False],
 61: (4)                 ],
 62: (4)                 [
 63: (8)                     ["^mine$", "ours", None, False],
 64: (8)                     ["^yours$|^thine$", "yours", None, False],
 65: (8)                     ["^hers$|^his$|^its$|^theirs$", "theirs", None, False],
 66: (4)                 ],
 67: (4)                 [
 68: (8)                     ["^I$", "we", None, False],
 69: (8)                     ["^me$", "us", None, False],
 70: (8)                     ["^myself$", "ourselves", None, False],
 71: (8)                     ["^you$", "you", None, False],
 72: (8)                     ["^thou$|^thee$", "ye", None, False],
 73: (8)                     ["^yourself$|^thyself$", "yourself", None, False],
 74: (8)                     ["^she$|^he$|^it$|^they$", "they", None, False],
 75: (8)                     ["^her$|^him$|^it$|^them$", "them", None, False],
 76: (8)                     ["^herself$|^himself$|^itself$|^themself$", "themselves", None,
False],
 77: (8)                     ["^oneself$", "oneselves", None, False],
 78: (4)                 ],
 79: (4)                 [
 80: (8)                     ["$", "", "uninflected", False],
 81: (8)                     ["$", "", "uncountable", False],
 82: (8)                     ["fish$", "fish", None, False],
 83: (8)                     ["([- ])bass$", "\\1bass", None, False],
 84: (8)                     ["ois$", "ois", None, False],
 85: (8)                     ["sheep$", "sheep", None, False],
 86: (8)                     ["deer$", "deer", None, False],
 87: (8)                     ["pox$", "pox", None, False],
 88: (8)                     ["([A-Z].*)ese$", "\\1ese", None, False],
 89: (8)                     ["itis$", "itis", None, False],
 90: (8)                     [
 91: (12)                        "(fruct|gluc|galact|lact|ket|malt|rib|sacchar|cellul)ose$",
 92: (12)                        "\\1ose",
 93: (12)                        None,
 94: (12)                        False,
 95: (8)                     ],
 96: (4)                 ],
 97: (4)                 [
 98: (8)                     ["atlas$", "atlantes", None, True],
 99: (8)                     ["atlas$", "atlases", None, False],
100: (8)                     ["beef$", "beeves", None, True],
101: (8)                     ["brother$", "brethren", None, True],
102: (8)                     ["child$", "children", None, False],
103: (8)                     ["corpus$", "corpora", None, True],
104: (8)                     ["corpus$", "corpuses", None, False],
105: (8)                     ["^cow$", "kine", None, True],
106: (8)                     ["ephemeris$", "ephemerides", None, False],
107: (8)                     ["ganglion$", "ganglia", None, True],
108: (8)                     ["genie$", "genii", None, True],
109: (8)                     ["genus$", "genera", None, False],
110: (8)                     ["graffito$", "graffiti", None, False],
```

```
111: (8)                     ["loaf$", "loaves", None, False],
112: (8)                     ["money$", "monies", None, True],
113: (8)                     ["mongoose$", "mongooses", None, False],
114: (8)                     ["mythos$", "mythoi", None, False],
115: (8)                     ["octopus$", "octopodes", None, True],
116: (8)                     ["opus$", "opera", None, True],
117: (8)                     ["opus$", "opuses", None, False],
118: (8)                     ["^ox$", "oxen", None, False],
119: (8)                     ["penis$", "penes", None, True],
120: (8)                     ["penis$", "penises", None, False],
121: (8)                     ["soliloquy$", "soliloquies", None, False],
122: (8)                     ["testis$", "testes", None, False],
123: (8)                     ["trilby$", "trilbys", None, False],
124: (8)                     ["turf$", "turves", None, True],
125: (8)                     ["numen$", "numena", None, False],
126: (8)                     ["occiput$", "occipita", None, True],
127: (4)                 ],
128: (4)                 [
129: (8)                     ["man$", "men", None, False],
130: (8)                     ["person$", "people", None, False],
131: (8)                     ["([lm])ouse$", "\\1ice", None, False],
132: (8)                     ["tooth$", "teeth", None, False],
133: (8)                     ["goose$", "geese", None, False],
134: (8)                     ["foot$", "feet", None, False],
135: (8)                     ["zoon$", "zoa", None, False],
136: (8)                     ["([csx])is$", "\\1es", None, False],
137: (4)                 ],
138: (4)                 [
139: (8)                     ["ex$", "ices", "ex-ices", False],
140: (8)                     ["ex$", "ices", "ex-ices-classical", True],
141: (8)                     ["um$", "a", "um-a", False],
142: (8)                     ["um$", "a", "um-a-classical", True],
143: (8)                     ["on$", "a", "on-a", False],
144: (8)                     ["a$", "ae", "a-ae", False],
145: (8)                     ["a$", "ae", "a-ae-classical", True],
146: (4)                 ],
147: (4)                 [
148: (8)                     ["trix$", "trices", None, True],
149: (8)                     ["eau$", "eaux", None, True],
150: (8)                     ["ieu$", "ieu", None, True],
151: (8)                     ["([iay])nx$", "\\1nges", None, True],
152: (8)                     ["en$", "ina", "en-ina-classical", True],
153: (8)                     ["a$", "ata", "a-ata-classical", True],
154: (8)                     ["is$", "ides", "is-ides-classical", True],
155: (8)                     ["us$", "i", "us-i-classical", True],
156: (8)                     ["us$", "us", "us-us-classical", True],
157: (8)                     ["o$", "i", "o-i-classical", True],
158: (8)                     ["$", "i", "-i-classical", True],
159: (8)                     ["$", "im", "-im-classical", True],
160: (4)                 ],
161: (4)                 [
162: (8)                     ["([cs])h$", "\\1hes", None, False],
163: (8)                     ["ss$", "sses", None, False],
164: (8)                     ["x$", "xes", None, False],
165: (8)                     ["s$", "ses", "s-singular", False],
166: (4)                 ],
167: (4)                 [
168: (8)                     ["([aeo]l)f$", "\\1ves", None, False],
169: (8)                     ["([^d]ea)f$", "\\1ves", None, False],
170: (8)                     ["arf$", "arves", None, False],
171: (8)                     ["([nlw]i)fe$", "\\1ves", None, False],
172: (4)                 ],
173: (4)                 [
174: (8)                     ["([aeiou])y$", "\\1ys", None, False],
175: (8)                     ["([A-Z].*)y$", "\\1ys", None, False],
176: (8)                     ["y$", "ies", None, False],
177: (4)                 ],
178: (4)                 [
179: (8)                     ["o$", "os", "o-os", False],
```

```
180: (8)                    ["([aeiou])o$", "\\1os", None, False],
181: (8)                    ["o$", "oes", None, False],
182: (4)                ],
183: (4)                [["l$", "ls", "general-generals", False]],
184: (4)                [["$", "s", None, False]],
185: (0)            ]
186: (0)            for ruleset in plural_rules:
187: (4)                for rule in ruleset:
188: (8)                    rule[0] = re.compile(rule[0])
189: (0)            plural_categories = {
190: (4)                "uninflected": [
191: (8)                    "aircraft",
192: (8)                    "antelope",
193: (8)                    "bison",
194: (8)                    "bream",
195: (8)                    "breeches",
196: (8)                    "britches",
197: (8)                    "carp",
198: (8)                    "cattle",
199: (8)                    "chassis",
200: (8)                    "clippers",
201: (8)                    "cod",
202: (8)                    "contretemps",
203: (8)                    "corps",
204: (8)                    "debris",
205: (8)                    "diabetes",
206: (8)                    "djinn",
207: (8)                    "eland",
208: (8)                    "elk",
209: (8)                    "flounder",
210: (8)                    "gallows",
211: (8)                    "graffiti",
212: (8)                    "headquarters",
213: (8)                    "herpes",
214: (8)                    "high-jinks",
215: (8)                    "homework",
216: (8)                    "innings",
217: (8)                    "jackanapes",
218: (8)                    "mackerel",
219: (8)                    "measles",
220: (8)                    "mews",
221: (8)                    "moose",
222: (8)                    "mumps",
223: (8)                    "offspring",
224: (8)                    "news",
225: (8)                    "pincers",
226: (8)                    "pliers",
227: (8)                    "proceedings",
228: (8)                    "rabies",
229: (8)                    "salmon",
230: (8)                    "scissors",
231: (8)                    "series",
232: (8)                    "shears",
233: (8)                    "species",
234: (8)                    "swine",
235: (8)                    "trout",
236: (8)                    "tuna",
237: (8)                    "whiting",
238: (8)                    "wildebeest",
239: (4)                ],
240: (4)                "uncountable": [
241: (8)                    "advice",
242: (8)                    "bread",
243: (8)                    "butter",
244: (8)                    "cannabis",
245: (8)                    "cheese",
246: (8)                    "electricity",
247: (8)                    "equipment",
248: (8)                    "fruit",
```

```
249: (8)                    "furniture",
250: (8)                    "garbage",
251: (8)                    "gravel",
252: (8)                    "happiness",
253: (8)                    "information",
254: (8)                    "ketchup",
255: (8)                    "knowledge",
256: (8)                    "love",
257: (8)                    "luggage",
258: (8)                    "mathematics",
259: (8)                    "mayonnaise",
260: (8)                    "meat",
261: (8)                    "mustard",
262: (8)                    "news",
263: (8)                    "progress",
264: (8)                    "research",
265: (8)                    "rice",
266: (8)                    "sand",
267: (8)                    "software",
268: (8)                    "understanding",
269: (8)                    "water",
270: (4)                ],
271: (4)                "s-singular": [
272: (8)                    "acropolis",
273: (8)                    "aegis",
274: (8)                    "alias",
275: (8)                    "asbestos",
276: (8)                    "bathos",
277: (8)                    "bias",
278: (8)                    "bus",
279: (8)                    "caddis",
280: (8)                    "canvas",
281: (8)                    "chaos",
282: (8)                    "christmas",
283: (8)                    "cosmos",
284: (8)                    "dais",
285: (8)                    "digitalis",
286: (8)                    "epidermis",
287: (8)                    "ethos",
288: (8)                    "gas",
289: (8)                    "glottis",
290: (8)                    "ibis",
291: (8)                    "lens",
292: (8)                    "mantis",
293: (8)                    "marquis",
294: (8)                    "metropolis",
295: (8)                    "pathos",
296: (8)                    "pelvis",
297: (8)                    "polis",
298: (8)                    "rhinoceros",
299: (8)                    "sassafras",
300: (8)                    "trellis",
301: (4)                ],
302: (4)                "ex-ices": ["codex", "murex", "silex"],
303: (4)                "ex-ices-classical": [
304: (8)                    "apex",
305: (8)                    "cortex",
306: (8)                    "index",
307: (8)                    "latex",
308: (8)                    "pontifex",
309: (8)                    "simplex",
310: (8)                    "vertex",
311: (8)                    "vortex",
312: (4)                ],
313: (4)                "um-a": [
314: (8)                    "agendum",
315: (8)                    "bacterium",
316: (8)                    "candelabrum",
317: (8)                    "datum",
```

```
318: (8)                 "desideratum",
319: (8)                 "erratum",
320: (8)                 "extremum",
321: (8)                 "ovum",
322: (8)                 "stratum",
323: (4)             ],
324: (4)             "um-a-classical": [
325: (8)                 "aquarium",
326: (8)                 "compendium",
327: (8)                 "consortium",
328: (8)                 "cranium",
329: (8)                 "curriculum",
330: (8)                 "dictum",
331: (8)                 "emporium",
332: (8)                 "enconium",
333: (8)                 "gymnasium",
334: (8)                 "honorarium",
335: (8)                 "interregnum",
336: (8)                 "lustrum",
337: (8)                 "maximum",
338: (8)                 "medium",
339: (8)                 "memorandum",
340: (8)                 "millenium",
341: (8)                 "minimum",
342: (8)                 "momentum",
343: (8)                 "optimum",
344: (8)                 "phylum",
345: (8)                 "quantum",
346: (8)                 "rostrum",
347: (8)                 "spectrum",
348: (8)                 "speculum",
349: (8)                 "stadium",
350: (8)                 "trapezium",
351: (8)                 "ultimatum",
352: (8)                 "vacuum",
353: (8)                 "velum",
354: (4)             ],
355: (4)             "on-a": [
356: (8)                 "aphelion",
357: (8)                 "asyndeton",
358: (8)                 "criterion",
359: (8)                 "hyperbaton",
360: (8)                 "noumenon",
361: (8)                 "organon",
362: (8)                 "perihelion",
363: (8)                 "phenomenon",
364: (8)                 "prolegomenon",
365: (4)             ],
366: (4)             "a-ae": ["alga", "alumna", "vertebra"],
367: (4)             "a-ae-classical": [
368: (8)                 "abscissa",
369: (8)                 "amoeba",
370: (8)                 "antenna",
371: (8)                 "aurora",
372: (8)                 "formula",
373: (8)                 "hydra",
374: (8)                 "hyperbola",
375: (8)                 "lacuna",
376: (8)                 "medusa",
377: (8)                 "nebula",
378: (8)                 "nova",
379: (8)                 "parabola",
380: (4)             ],
381: (4)             "en-ina-classical": ["foramen", "lumen", "stamen"],
382: (4)             "a-ata-classical": [
383: (8)                 "anathema",
384: (8)                 "bema",
385: (8)                 "carcinoma",
386: (8)                 "charisma",
```

```
387: (8)                    "diploma",
388: (8)                    "dogma",
389: (8)                    "drama",
390: (8)                    "edema",
391: (8)                    "enema",
392: (8)                    "enigma",
393: (8)                    "gumma",
394: (8)                    "lemma",
395: (8)                    "lymphoma",
396: (8)                    "magma",
397: (8)                    "melisma",
398: (8)                    "miasma",
399: (8)                    "oedema",
400: (8)                    "sarcoma",
401: (8)                    "schema",
402: (8)                    "soma",
403: (8)                    "stigma",
404: (8)                    "stoma",
405: (8)                    "trauma",
406: (4)                ],
407: (4)                "is-ides-classical": ["clitoris", "iris"],
408: (4)                "us-i-classical": [
409: (8)                    "focus",
410: (8)                    "fungus",
411: (8)                    "genius",
412: (8)                    "incubus",
413: (8)                    "nimbus",
414: (8)                    "nucleolus",
415: (8)                    "radius",
416: (8)                    "stylus",
417: (8)                    "succubus",
418: (8)                    "torus",
419: (8)                    "umbilicus",
420: (8)                    "uterus",
421: (4)                ],
422: (4)                "us-us-classical": [
423: (8)                    "apparatus",
424: (8)                    "cantus",
425: (8)                    "coitus",
426: (8)                    "hiatus",
427: (8)                    "impetus",
428: (8)                    "nexus",
429: (8)                    "plexus",
430: (8)                    "prospectus",
431: (8)                    "sinus",
432: (8)                    "status",
433: (4)                ],
434: (4)                "o-i-classical": [
435: (8)                    "alto",
436: (8)                    "basso",
437: (8)                    "canto",
438: (8)                    "contralto",
439: (8)                    "crescendo",
440: (8)                    "solo",
441: (8)                    "soprano",
442: (8)                    "tempo",
443: (4)                ],
444: (4)                "-i-classical": ["afreet", "afrit", "efreet"],
445: (4)                "-im-classical": ["cherub", "goy", "seraph"],
446: (4)                "o-os": [
447: (8)                    "albino",
448: (8)                    "archipelago",
449: (8)                    "armadillo",
450: (8)                    "commando",
451: (8)                    "ditto",
452: (8)                    "dynamo",
453: (8)                    "embryo",
454: (8)                    "fiasco",
455: (8)                    "generalissimo",
```

```
456: (8)                    "ghetto",
457: (8)                    "guano",
458: (8)                    "inferno",
459: (8)                    "jumbo",
460: (8)                    "lingo",
461: (8)                    "lumbago",
462: (8)                    "magneto",
463: (8)                    "manifesto",
464: (8)                    "medico",
465: (8)                    "octavo",
466: (8)                    "photo",
467: (8)                    "pro",
468: (8)                    "quarto",
469: (8)                    "rhino",
470: (8)                    "stylo",
471: (4)                ],
472: (4)                "general-generals": [
473: (8)                    "Adjutant",
474: (8)                    "Brigadier",
475: (8)                    "Lieutenant",
476: (8)                    "Major",
477: (8)                    "Quartermaster",
478: (8)                    "adjutant",
479: (8)                    "brigadier",
480: (8)                    "lieutenant",
481: (8)                    "major",
482: (8)                    "quartermaster",
483: (4)                ],
484: (0)            }
485: (0)            def pluralize(word, pos=NOUN, custom=None, classical=True):
486: (4)                """Returns the plural of a given word.
487: (4)                For example: child -> children.
488: (4)                Handles nouns and adjectives, using classical inflection by default
489: (4)                (e.g. where "matrix" pluralizes to "matrices" instead of "matrixes").
490: (4)                The custom dictionary is for user-defined replacements.
491: (4)                """
492: (4)                if custom is None:
493: (8)                    custom = {}
494: (4)                if word in custom:
495: (8)                    return custom[word]
496: (4)                if word.endswith("'") or word.endswith("'s"):
497: (8)                    owner = word.rstrip("'s")
498: (8)                    owners = pluralize(owner, pos, custom, classical)
499: (8)                    if owners.endswith("s"):
500: (12)                       return owners + "'"
501: (8)                    else:
502: (12)                       return owners + "'s"
503: (4)                words = word.replace("-", " ").split(" ")
504: (4)                if len(words) > 1:
505: (8)                    if (
506: (12)                       words[1] == "general"
507: (12)                       or words[1] == "General"
508: (12)                       and words[0] not in plural_categories["general-generals"]
509: (8)                    ):
510: (12)                       return word.replace(words[0], pluralize(words[0], pos, custom,
classical))
511: (8)                    elif words[1] in plural_prepositions:
512: (12)                       return word.replace(words[0], pluralize(words[0], pos, custom,
classical))
513: (8)                    else:
514: (12)                       return word.replace(words[-1], pluralize(words[-1], pos, custom,
classical))
515: (4)                n = list(range(len(plural_rules)))
516: (4)                if pos.startswith(ADJECTIVE):
517: (8)                    n = [0, 1]
518: (4)                for i in n:
519: (8)                    ruleset = plural_rules[i]
520: (8)                    for rule in ruleset:
521: (12)                       suffix, inflection, category, classic = rule
```

```
522: (12)                         if category is None:
523: (16)                             if not classic or (classic and classical):
524: (20)                                 if suffix.search(word) is not None:
525: (24)                                     return suffix.sub(inflection, word)
526: (12)                         if category is not None:
527: (16)                             if word in plural_categories[category] and (
528: (20)                                 not classic or (classic and classical)
529: (16)                             ):
530: (20)                                 if suffix.search(word) is not None:
531: (24)                                     return suffix.sub(inflection, word)
532: (0)              singular_rules = [
533: (4)                  ["(?i)(.)ae$", "\\1a"],
534: (4)                  ["(?i)(.)itis$", "\\1itis"],
535: (4)                  ["(?i)(.)eaux$", "\\1eau"],
536: (4)                  ["(?i)(quiz)zes$", "\\1"],
537: (4)                  ["(?i)(matr)ices$", "\\1ix"],
538: (4)                  ["(?i)(ap|vert|ind)ices$", "\\1ex"],
539: (4)                  ["(?i)^(ox)en", "\\1"],
540: (4)                  ["(?i)(alias|status)es$", "\\1"],
541: (4)                  ["(?i)([octop|vir])i$", "\\1us"],
542: (4)                  ["(?i)(cris|ax|test)es$", "\\1is"],
543: (4)                  ["(?i)(shoe)s$", "\\1"],
544: (4)                  ["(?i)(o)es$", "\\1"],
545: (4)                  ["(?i)(bus)es$", "\\1"],
546: (4)                  ["(?i)([m|l])ice$", "\\1ouse"],
547: (4)                  ["(?i)(x|ch|ss|sh)es$", "\\1"],
548: (4)                  ["(?i)(m)ovies$", "\\1ovie"],
549: (4)                  ["(?i)(.)ombies$", "\\1ombie"],
550: (4)                  ["(?i)(s)eries$", "\\1eries"],
551: (4)                  ["(?i)([^aeiouy]|qu)ies$", "\\1y"],
552: (4)                  ["([aeo]l)ves$", "\\1f"],
553: (4)                  ["([^d]ea)ves$", "\\1f"],
554: (4)                  ["arves$", "arf"],
555: (4)                  ["erves$", "erve"],
556: (4)                  ["([nlw]i)ves$", "\\1fe"],
557: (4)                  ["(?i)([lr])ves$", "\\1f"],
558: (4)                  ["([aeo])ves$", "\\1ve"],
559: (4)                  ["(?i)(sive)s$", "\\1"],
560: (4)                  ["(?i)(tive)s$", "\\1"],
561: (4)                  ["(?i)(hive)s$", "\\1"],
562: (4)                  ["(?i)([^f])ves$", "\\1fe"],
563: (4)                  ["(?i)(^analy)ses$", "\\1sis"],
564: (4)                  ["(?i)((a)naly|(b)a|(d)iagno|(p)arenthe|(p)rogno|(s)ynop|(t)he)ses$",
"\\1\\2sis"],
565: (4)                  ["(?i)(.)opses$", "\\1opsis"],
566: (4)                  ["(?i)(.)yses$", "\\1ysis"],
567: (4)                  ["(?i)(h|d|r|o|n|b|cl|p)oses$", "\\1ose"],
568: (4)                  ["(?i)(fruct|gluc|galact|lact|ket|malt|rib|sacchar|cellul)ose$",
"\\1ose"],
569: (4)                  ["(?i)(.)oses$", "\\1osis"],
570: (4)                  ["(?i)([ti])a$", "\\1um"],
571: (4)                  ["(?i)(n)ews$", "\\1ews"],
572: (4)                  ["(?i)s$", ""],
573: (0)              ]
574: (0)              for rule in singular_rules:
575: (4)                  rule[0] = re.compile(rule[0])
576: (0)              singular_uninflected = [
577: (4)                  "aircraft",
578: (4)                  "antelope",
579: (4)                  "bison",
580: (4)                  "bream",
581: (4)                  "breeches",
582: (4)                  "britches",
583: (4)                  "carp",
584: (4)                  "cattle",
585: (4)                  "chassis",
586: (4)                  "clippers",
587: (4)                  "cod",
588: (4)                  "contretemps",
```

```
589: (4)                "corps",
590: (4)                "debris",
591: (4)                "diabetes",
592: (4)                "djinn",
593: (4)                "eland",
594: (4)                "elk",
595: (4)                "flounder",
596: (4)                "gallows",
597: (4)                "georgia",
598: (4)                "graffiti",
599: (4)                "headquarters",
600: (4)                "herpes",
601: (4)                "high-jinks",
602: (4)                "homework",
603: (4)                "innings",
604: (4)                "jackanapes",
605: (4)                "mackerel",
606: (4)                "measles",
607: (4)                "mews",
608: (4)                "moose",
609: (4)                "mumps",
610: (4)                "news",
611: (4)                "offspring",
612: (4)                "pincers",
613: (4)                "pliers",
614: (4)                "proceedings",
615: (4)                "rabies",
616: (4)                "salmon",
617: (4)                "scissors",
618: (4)                "series",
619: (4)                "shears",
620: (4)                "species",
621: (4)                "swine",
622: (4)                "swiss",
623: (4)                "trout",
624: (4)                "tuna",
625: (4)                "whiting",
626: (4)                "wildebeest",
627: (0)            ]
628: (0)            singular_uncountable = [
629: (4)                "advice",
630: (4)                "bread",
631: (4)                "butter",
632: (4)                "cannabis",
633: (4)                "cheese",
634: (4)                "electricity",
635: (4)                "equipment",
636: (4)                "fruit",
637: (4)                "furniture",
638: (4)                "garbage",
639: (4)                "gravel",
640: (4)                "happiness",
641: (4)                "information",
642: (4)                "ketchup",
643: (4)                "knowledge",
644: (4)                "love",
645: (4)                "luggage",
646: (4)                "mathematics",
647: (4)                "mayonnaise",
648: (4)                "meat",
649: (4)                "mustard",
650: (4)                "news",
651: (4)                "progress",
652: (4)                "research",
653: (4)                "rice",
654: (4)                "sand",
655: (4)                "software",
656: (4)                "understanding",
657: (4)                "water",
```

```
658: (0)            ]
659: (0)            singular_ie = [
660: (4)                "algerie",
661: (4)                "auntie",
662: (4)                "beanie",
663: (4)                "birdie",
664: (4)                "bogie",
665: (4)                "bombie",
666: (4)                "bookie",
667: (4)                "collie",
668: (4)                "cookie",
669: (4)                "cutie",
670: (4)                "doggie",
671: (4)                "eyrie",
672: (4)                "freebie",
673: (4)                "goonie",
674: (4)                "groupie",
675: (4)                "hankie",
676: (4)                "hippie",
677: (4)                "hoagie",
678: (4)                "hottie",
679: (4)                "indie",
680: (4)                "junkie",
681: (4)                "laddie",
682: (4)                "laramie",
683: (4)                "lingerie",
684: (4)                "meanie",
685: (4)                "nightie",
686: (4)                "oldie",
687: (4)                "^pie",
688: (4)                "pixie",
689: (4)                "quickie",
690: (4)                "reverie",
691: (4)                "rookie",
692: (4)                "softie",
693: (4)                "sortie",
694: (4)                "stoolie",
695: (4)                "sweetie",
696: (4)                "techie",
697: (4)                "^tie",
698: (4)                "toughie",
699: (4)                "valkyrie",
700: (4)                "veggie",
701: (4)                "weenie",
702: (4)                "yuppie",
703: (4)                "zombie",
704: (0)            ]
705: (0)            singular_s = plural_categories["s-singular"]
706: (0)            singular_irregular = {
707: (4)                "men": "man",
708: (4)                "people": "person",
709: (4)                "children": "child",
710: (4)                "sexes": "sex",
711: (4)                "axes": "axe",
712: (4)                "moves": "move",
713: (4)                "teeth": "tooth",
714: (4)                "geese": "goose",
715: (4)                "feet": "foot",
716: (4)                "zoa": "zoon",
717: (4)                "atlantes": "atlas",
718: (4)                "atlases": "atlas",
719: (4)                "beeves": "beef",
720: (4)                "brethren": "brother",
721: (4)                "corpora": "corpus",
722: (4)                "corpuses": "corpus",
723: (4)                "kine": "cow",
724: (4)                "ephemerides": "ephemeris",
725: (4)                "ganglia": "ganglion",
726: (4)                "genii": "genie",
```

```
727: (4)                "genera": "genus",
728: (4)                "graffiti": "graffito",
729: (4)                "helves": "helve",
730: (4)                "leaves": "leaf",
731: (4)                "loaves": "loaf",
732: (4)                "monies": "money",
733: (4)                "mongooses": "mongoose",
734: (4)                "mythoi": "mythos",
735: (4)                "octopodes": "octopus",
736: (4)                "opera": "opus",
737: (4)                "opuses": "opus",
738: (4)                "oxen": "ox",
739: (4)                "penes": "penis",
740: (4)                "penises": "penis",
741: (4)                "soliloquies": "soliloquy",
742: (4)                "testes": "testis",
743: (4)                "trilbys": "trilby",
744: (4)                "turves": "turf",
745: (4)                "numena": "numen",
746: (4)                "occipita": "occiput",
747: (4)                "our": "my",
748: (0)            }
749: (0)            def singularize(word, pos=NOUN, custom=None):
750: (4)                if custom is None:
751: (8)                    custom = {}
752: (4)                if word in list(custom.keys()):
753: (8)                    return custom[word]
754: (4)                if "-" in word:
755: (8)                    words = word.split("-")
756: (8)                    if len(words) > 1 and words[1] in plural_prepositions:
757: (12)                       return singularize(words[0], pos, custom) + "-" + "-
".join(words[1:])
758: (4)                if word.endswith("'"):
759: (8)                    return singularize(word[:-1]) + "'s"
760: (4)                lower = word.lower()
761: (4)                for w in singular_uninflected:
762: (8)                    if w.endswith(lower):
763: (12)                       return word
764: (4)                for w in singular_uncountable:
765: (8)                    if w.endswith(lower):
766: (12)                       return word
767: (4)                for w in singular_ie:
768: (8)                    if lower.endswith(w + "s"):
769: (12)                       return w
770: (4)                for w in singular_s:
771: (8)                    if lower.endswith(w + "es"):
772: (12)                       return w
773: (4)                for w in list(singular_irregular.keys()):
774: (8)                    if lower.endswith(w):
775: (12)                       return re.sub("(?i)" + w + "$", singular_irregular[w], word)
776: (4)                for rule in singular_rules:
777: (8)                    suffix, inflection = rule
778: (8)                    match = suffix.search(word)
779: (8)                    if match:
780: (12)                       groups = match.groups()
781: (12)                       for k in range(0, len(groups)):
782: (16)                           if groups[k] is None:
783: (20)                               inflection = inflection.replace("\\" + str(k + 1), "")
784: (12)                       return suffix.sub(inflection, word)
785: (4)                return word
```

File 21 - np_extractors.py:

```
1: (0)              """Various noun phrase extractors."""
2: (0)              import nltk
3: (0)              from textblob.base import BaseNPExtractor
4: (0)              from textblob.decorators import requires_nltk_corpus
```

```
5: (0)              from textblob.taggers import PatternTagger
6: (0)              from textblob.utils import filter_insignificant, tree2str
7: (0)              class ChunkParser(nltk.ChunkParserI):
8: (4)                  def __init__(self):
9: (8)                      self._trained = False
10: (4)                 @requires_nltk_corpus
11: (4)                 def train(self):
12: (8)                     """Train the Chunker on the ConLL-2000 corpus."""
13: (8)                     train_data = [
14: (12)                        [(t, c) for _, t, c in nltk.chunk.tree2conlltags(sent)]
15: (12)                        for sent in nltk.corpus.conll2000.chunked_sents(
16: (16)                            "train.txt", chunk_types=["NP"]
17: (12)                        )
18: (8)                     ]
19: (8)                     unigram_tagger = nltk.UnigramTagger(train_data)
20: (8)                     self.tagger = nltk.BigramTagger(train_data, backoff=unigram_tagger)
21: (8)                     self._trained = True
22: (4)                 def parse(self, sentence):
23: (8)                     """Return the parse tree for the sentence."""
24: (8)                     if not self._trained:
25: (12)                        self.train()
26: (8)                     pos_tags = [pos for (word, pos) in sentence]
27: (8)                     tagged_pos_tags = self.tagger.tag(pos_tags)
28: (8)                     chunktags = [chunktag for (pos, chunktag) in tagged_pos_tags]
29: (8)                     conlltags = [
30: (12)                        (word, pos, chunktag)
31: (12)                        for ((word, pos), chunktag) in zip(sentence, chunktags)
32: (8)                     ]
33: (8)                     return nltk.chunk.util.conlltags2tree(conlltags)
34: (0)             class ConllExtractor(BaseNPExtractor):
35: (4)                 """A noun phrase extractor that uses chunk parsing trained with the
36: (4)                 ConLL-2000 training corpus.
37: (4)                 """
38: (4)                 POS_TAGGER = PatternTagger()
39: (4)                 CFG = {
40: (8)                     ("NNP", "NNP"): "NNP",
41: (8)                     ("NN", "NN"): "NNI",
42: (8)                     ("NNI", "NN"): "NNI",
43: (8)                     ("JJ", "JJ"): "JJ",
44: (8)                     ("JJ", "NN"): "NNI",
45: (4)                 }
46: (4)                 INSIGNIFICANT_SUFFIXES = ["DT", "CC", "PRP$", "PRP"]
47: (4)                 def __init__(self, parser=None):
48: (8)                     self.parser = ChunkParser() if not parser else parser
49: (4)                 def extract(self, text):
50: (8)                     """Return a list of noun phrases (strings) for body of text."""
51: (8)                     sentences = nltk.tokenize.sent_tokenize(text)
52: (8)                     noun_phrases = []
53: (8)                     for sentence in sentences:
54: (12)                        parsed = self._parse_sentence(sentence)
55: (12)                        phrases = [
56: (16)                            _normalize_tags(filter_insignificant(each,
self.INSIGNIFICANT_SUFFIXES))
57: (16)                            for each in parsed
58: (16)                            if isinstance(each, nltk.tree.Tree)
59: (16)                            and each.label() == "NP"
60: (16)                            and len(filter_insignificant(each)) >= 1
61: (16)                            and _is_match(each, cfg=self.CFG)
62: (12)                        ]
63: (12)                        nps = [tree2str(phrase) for phrase in phrases]
64: (12)                        noun_phrases.extend(nps)
65: (8)                     return noun_phrases
66: (4)                 def _parse_sentence(self, sentence):
67: (8)                     """Tag and parse a sentence (a plain, untagged string)."""
68: (8)                     tagged = self.POS_TAGGER.tag(sentence)
69: (8)                     return self.parser.parse(tagged)
70: (0)             class FastNPExtractor(BaseNPExtractor):
71: (4)                 """A fast and simple noun phrase extractor.
72: (4)                 Credit to Shlomi Babluk. Link to original blog post:
```

```
                    http://thetokenizer.com/2013/05/09/efficient-way-to-extract-the-main-topics-of-a-sentence/
                """
                CFG = {
                    ("NNP", "NNP"): "NNP",
                    ("NN", "NN"): "NNI",
                    ("NNI", "NN"): "NNI",
                    ("JJ", "JJ"): "JJ",
                    ("JJ", "NN"): "NNI",
                }
                def __init__(self):
                    self._trained = False
                @requires_nltk_corpus
                def train(self):
                    train_data = nltk.corpus.brown.tagged_sents(categories="news")
                    regexp_tagger = nltk.RegexpTagger(
                        [
                            (r"^-?[0-9]+(.[0-9]+)?$", "CD"),
                            (r"(-|:|;)$", ":"),
                            (r"\'*$", "MD"),
                            (r"(The|the|A|a|An|an)$", "AT"),
                            (r".*able$", "JJ"),
                            (r"^[A-Z].*$", "NNP"),
                            (r".*ness$", "NN"),
                            (r".*ly$", "RB"),
                            (r".*s$", "NNS"),
                            (r".*ing$", "VBG"),
                            (r".*ed$", "VBD"),
                            (r".*", "NN"),
                        ]
                    )
                    unigram_tagger = nltk.UnigramTagger(train_data, backoff=regexp_tagger)
                    self.tagger = nltk.BigramTagger(train_data, backoff=unigram_tagger)
                    self._trained = True
                    return None
                def _tokenize_sentence(self, sentence):
                    """Split the sentence into single words/tokens"""
                    tokens = nltk.word_tokenize(sentence)
                    return tokens
                def extract(self, sentence):
                    """Return a list of noun phrases (strings) for body of text."""
                    if not self._trained:
                        self.train()
                    tokens = self._tokenize_sentence(sentence)
                    tagged = self.tagger.tag(tokens)
                    tags = _normalize_tags(tagged)
                    merge = True
                    while merge:
                        merge = False
                        for x in range(0, len(tags) - 1):
                            t1 = tags[x]
                            t2 = tags[x + 1]
                            key = t1[1], t2[1]
                            value = self.CFG.get(key, "")
                            if value:
                                merge = True
                                tags.pop(x)
                                tags.pop(x)
                                match = f"{t1[0]} {t2[0]}"
                                pos = value
                                tags.insert(x, (match, pos))
                                break
                    matches = [t[0] for t in tags if t[1] in ["NNP", "NNI"]]
                    return matches
            def _normalize_tags(chunk):
                """Normalize the corpus tags.
                ("NN", "NN-PL", "NNS") -> "NN"
                """
                ret = []
```

```
141: (4)                for word, tag in chunk:
142: (8)                    if tag == "NP-TL" or tag == "NP":
143: (12)                       ret.append((word, "NNP"))
144: (12)                       continue
145: (8)                    if tag.endswith("-TL"):
146: (12)                       ret.append((word, tag[:-3]))
147: (12)                       continue
148: (8)                    if tag.endswith("S"):
149: (12)                       ret.append((word, tag[:-1]))
150: (12)                       continue
151: (8)                    ret.append((word, tag))
152: (4)                return ret
153: (0)            def _is_match(tagged_phrase, cfg):
154: (4)                """Return whether or not a tagged phrases matches a context-free
grammar."""
155: (4)                copy = list(tagged_phrase)  # A copy of the list
156: (4)                merge = True
157: (4)                while merge:
158: (8)                    merge = False
159: (8)                    for i in range(len(copy) - 1):
160: (12)                       first, second = copy[i], copy[i + 1]
161: (12)                       key = first[1], second[1]  # Tuple of tags e.g. ('NN', 'JJ')
162: (12)                       value = cfg.get(key, None)
163: (12)                       if value:
164: (16)                           merge = True
165: (16)                           copy.pop(i)
166: (16)                           copy.pop(i)
167: (16)                           match = f"{first[0]} {second[0]}"
168: (16)                           pos = value
169: (16)                           copy.insert(i, (match, pos))
170: (16)                           break
171: (4)                match = any([t[1] in ("NNP", "NNI") for t in copy])
172: (4)                return match
```

File 22 - parsers.py:

```
1: (0)              """Various parser implementations.
2: (0)              .. versionadded:: 0.6.0
3: (0)              """
4: (0)              from textblob.base import BaseParser
5: (0)              from textblob.en import parse as pattern_parse
6: (0)              class PatternParser(BaseParser):
7: (4)                  """Parser that uses the implementation in Tom de Smedt's pattern library.
8: (4)                  http://www.clips.ua.ac.be/pages/pattern-en#parser
9: (4)                  """
10: (4)                 def parse(self, text):
11: (8)                     """Parses the text."""
12: (8)                     return pattern_parse(text)
```

File 23 - taggers.py:

```
1: (0)              """Parts-of-speech tagger implementations."""
2: (0)              import nltk
3: (0)              import textblob as tb
4: (0)              from textblob.base import BaseTagger
5: (0)              from textblob.decorators import requires_nltk_corpus
6: (0)              from textblob.en import tag as pattern_tag
7: (0)              class PatternTagger(BaseTagger):
8: (4)                  """Tagger that uses the implementation in
9: (4)                  Tom de Smedt's pattern library
10: (4)                 (http://www.clips.ua.ac.be/pattern).
11: (4)                 """
12: (4)                 def tag(self, text, tokenize=True):
13: (8)                     """Tag a string or BaseBlob."""
14: (8)                     if not isinstance(text, str):
```

```
15: (12)                        text = text.raw
16: (8)                     return pattern_tag(text, tokenize)
17: (0)             class NLTKTagger(BaseTagger):
18: (4)                 """Tagger that uses NLTK's standard TreeBank tagger.
19: (4)                 NOTE: Requires numpy. Not yet supported with PyPy.
20: (4)                 """
21: (4)                 @requires_nltk_corpus
22: (4)                 def tag(self, text):
23: (8)                     """Tag a string or BaseBlob."""
24: (8)                     if isinstance(text, str):
25: (12)                        text = tb.TextBlob(text)
26: (8)                     return nltk.tag.pos_tag(text.tokens)
```

File 24 - __init__.py:

```
1: (0)              import csv
2: (0)              VERSION = (0, 9, 4)
3: (0)              __version__ = ".".join(map(str, VERSION))
4: (0)              pass_throughs = [
5: (4)                  "register_dialect",
6: (4)                  "unregister_dialect",
7: (4)                  "get_dialect",
8: (4)                  "list_dialects",
9: (4)                  "field_size_limit",
10: (4)                 "Dialect",
11: (4)                 "excel",
12: (4)                 "excel_tab",
13: (4)                 "Sniffer",
14: (4)                 "QUOTE_ALL",
15: (4)                 "QUOTE_MINIMAL",
16: (4)                 "QUOTE_NONNUMERIC",
17: (4)                 "QUOTE_NONE",
18: (4)                 "Error",
19: (0)             ]
20: (0)             __all__ = [
21: (4)                 "reader",
22: (4)                 "writer",
23: (4)                 "DictReader",
24: (4)                 "DictWriter",
25: (0)             ] + pass_throughs
26: (0)             for prop in pass_throughs:
27: (4)                 globals()[prop] = getattr(csv, prop)
28: (0)             def _stringify(s, encoding, errors):
29: (4)                 if s is None:
30: (8)                     return ""
31: (4)                 if isinstance(s, unicode):
32: (8)                     return s.encode(encoding, errors)
33: (4)                 elif isinstance(s, (int, float)):
34: (8)                     pass  # let csv.QUOTE_NONNUMERIC do its thing.
35: (4)                 elif not isinstance(s, str):
36: (8)                     s = str(s)
37: (4)                 return s
38: (0)             def _stringify_list(l, encoding, errors="strict"):
39: (4)                 try:
40: (8)                     return [_stringify(s, encoding, errors) for s in iter(l)]
41: (4)                 except TypeError as e:
42: (8)                     raise csv.Error(str(e))
43: (0)             def _unicodify(s, encoding):
44: (4)                 if s is None:
45: (8)                     return None
46: (4)                 if isinstance(s, (unicode, int, float)):
47: (8)                     return s
48: (4)                 elif isinstance(s, str):
49: (8)                     return s.decode(encoding)
50: (4)                 return s
51: (0)             class UnicodeWriter:
52: (4)                 """
```

```
53: (4)                 >>> import unicodecsv
54: (4)                 >>> from cStringIO import StringIO
55: (4)                 >>> f = StringIO()
56: (4)                 >>> w = unicodecsv.writer(f, encoding='utf-8')
57: (4)                 >>> w.writerow((u'é', u'ñ'))
58: (4)                 >>> f.seek(0)
59: (4)                 >>> r = unicodecsv.reader(f, encoding='utf-8')
60: (4)                 >>> row = r.next()
61: (4)                 >>> row[0] == u'é'
62: (4)                 True
63: (4)                 >>> row[1] == u'ñ'
64: (4)                 True
65: (4)                 """
66: (4)                 def __init__(
67: (8)                     self, f, dialect=csv.excel, encoding="utf-8", errors="strict", *args,
**kwds
68: (4)                 ):
69: (8)                     self.encoding = encoding
70: (8)                     self.writer = csv.writer(f, dialect, *args, **kwds)
71: (8)                     self.encoding_errors = errors
72: (4)                 def writerow(self, row):
73: (8)                     self.writer.writerow(_stringify_list(row, self.encoding,
self.encoding_errors))
74: (4)                 def writerows(self, rows):
75: (8)                     for row in rows:
76: (12)                        self.writerow(row)
77: (4)                 @property
78: (4)                 def dialect(self):
79: (8)                     return self.writer.dialect
80: (0)             writer = UnicodeWriter
81: (0)             class UnicodeReader:
82: (4)                 def __init__(self, f, dialect=None, encoding="utf-8", errors="strict",
**kwds):
83: (8)                     format_params = [
84: (12)                        "delimiter",
85: (12)                        "doublequote",
86: (12)                        "escapechar",
87: (12)                        "lineterminator",
88: (12)                        "quotechar",
89: (12)                        "quoting",
90: (12)                        "skipinitialspace",
91: (8)                     ]
92: (8)                     if dialect is None:
93: (12)                        if not any([kwd_name in format_params for kwd_name in
kwds.keys()]):
94: (16)                            dialect = csv.excel
95: (8)                     self.reader = csv.reader(f, dialect, **kwds)
96: (8)                     self.encoding = encoding
97: (8)                     self.encoding_errors = errors
98: (4)                 def next(self):
99: (8)                     row = self.reader.next()
100: (8)                    encoding = self.encoding
101: (8)                    encoding_errors = self.encoding_errors
102: (8)                    float_ = float
103: (8)                    unicode_ = unicode
104: (8)                    return [
105: (12)                       (
106: (16)                           value
107: (16)                           if isinstance(value, float_)
108: (16)                           else unicode_(value, encoding, encoding_errors)
109: (12)                       )
110: (12)                       for value in row
111: (8)                    ]
112: (4)                def __iter__(self):
113: (8)                    return self
114: (4)                @property
115: (4)                def dialect(self):
116: (8)                    return self.reader.dialect
117: (4)                @property
```

```
118: (4)                def line_num(self):
119: (8)                    return self.reader.line_num
120: (0)            reader = UnicodeReader
121: (0)            class DictWriter(csv.DictWriter):
122: (4)                """
123: (4)                >>> from cStringIO import StringIO
124: (4)                >>> f = StringIO()
125: (4)                >>> w = DictWriter(f, ['a', u'ñ', 'b'], restval=u'î')
126: (4)                >>> w.writerow({'a':'1', u'ñ':'2'})
127: (4)                >>> w.writerow({'a':'1', u'ñ':'2', 'b':u'ø'})
128: (4)                >>> w.writerow({'a':u'é', u'ñ':'2'})
129: (4)                >>> f.seek(0)
130: (4)                >>> r = DictReader(f, fieldnames=['a', u'ñ'], restkey='r')
131: (4)                >>> r.next() == {'a': u'1', u'ñ':'2', 'r': [u'î']}
132: (4)                True
133: (4)                >>> r.next() == {'a': u'1', u'ñ':'2', 'r': [u'\xc3\xb8']}
134: (4)                True
135: (4)                >>> r.next() == {'a': u'\xc3\xa9', u'ñ':'2', 'r': [u'\xc3\xae']}
136: (4)                True
137: (4)                """
138: (4)                def __init__(
139: (8)                    self,
140: (8)                    csvfile,
141: (8)                    fieldnames,
142: (8)                    restval="",
143: (8)                    extrasaction="raise",
144: (8)                    dialect="excel",
145: (8)                    encoding="utf-8",
146: (8)                    errors="strict",
147: (8)                    *args,
148: (8)                    **kwds,
149: (4)                ):
150: (8)                    self.encoding = encoding
151: (8)                    csv.DictWriter.__init__(
152: (12)                       self, csvfile, fieldnames, restval, extrasaction, dialect, *args,
**kwds
153: (8)                    )
154: (8)                    self.writer = UnicodeWriter(
155: (12)                       csvfile, dialect, encoding=encoding, errors=errors, *args, **kwds
156: (8)                    )
157: (8)                    self.encoding_errors = errors
158: (4)                def writeheader(self):
159: (8)                    _stringify_list(self.fieldnames, self.encoding, self.encoding_errors)
160: (8)                    header = dict(zip(self.fieldnames, self.fieldnames))
161: (8)                    self.writerow(header)
162: (0)            class DictReader(csv.DictReader):
163: (4)                """
164: (4)                >>> from cStringIO import StringIO
165: (4)                >>> f = StringIO()
166: (4)                >>> w = DictWriter(f, fieldnames=['name', 'place'])
167: (4)                >>> w.writerow({'name': 'Cary Grant', 'place': 'hollywood'})
168: (4)                >>> w.writerow({'name': 'Nathan Brillstone', 'place': u'øLand'})
169: (4)                >>> w.writerow({'name': u'Willam ø. Unicoder', 'place': u'éSpandland'})
170: (4)                >>> f.seek(0)
171: (4)                >>> r = DictReader(f, fieldnames=['name', 'place'])
172: (4)                >>> print r.next() == {'name': 'Cary Grant', 'place': 'hollywood'}
173: (4)                True
174: (4)                >>> print r.next() == {'name': 'Nathan Brillstone', 'place': u'øLand'}
175: (4)                True
176: (4)                >>> print r.next() == {'name': u'Willam ø. Unicoder', 'place':
u'éSpandland'}
177: (4)                True
178: (4)                """
179: (4)                def __init__(
180: (8)                    self,
181: (8)                    csvfile,
182: (8)                    fieldnames=None,
183: (8)                    restkey=None,
184: (8)                    restval=None,
```

```
185: (8)                    dialect="excel",
186: (8)                    encoding="utf-8",
187: (8)                    errors="strict",
188: (8)                    *args,
189: (8)                    **kwds,
190: (4)                ):
191: (8)                    if fieldnames is not None:
192: (12)                       fieldnames = _stringify_list(fieldnames, encoding)
193: (8)                    csv.DictReader.__init__(
194: (12)                       self, csvfile, fieldnames, restkey, restval, dialect, *args,
**kwds
195: (8)                    )
196: (8)                    self.reader = UnicodeReader(
197: (12)                       csvfile, dialect, encoding=encoding, errors=errors, *args, **kwds
198: (8)                    )
199: (8)                    if fieldnames is None and not hasattr(csv.DictReader, "fieldnames"):
200: (12)                       reader = UnicodeReader(csvfile, dialect, encoding=encoding, *args,
**kwds)
201: (12)                       self.fieldnames = _stringify_list(reader.next(), reader.encoding)
202: (8)                    self.unicode_fieldnames = [_unicodify(f, encoding) for f in
self.fieldnames]
203: (8)                    self.unicode_restkey = _unicodify(restkey, encoding)
204: (4)                def next(self):
205: (8)                    row = csv.DictReader.next(self)
206: (8)                    result = dict(
207: (12)                       (uni_key, row[str_key])
208: (12)                       for (str_key, uni_key) in zip(self.fieldnames,
self.unicode_fieldnames)
209: (8)                    )
210: (8)                    rest = row.get(self.restkey)
211: (8)                    if rest:
212: (12)                       result[self.unicode_restkey] = rest
213: (8)                    return result
```

File 25 - sentiments.py:

```
1: (0)              """Sentiment analysis implementations.
2: (0)              .. versionadded:: 0.5.0
3: (0)              """
4: (0)              from collections import namedtuple
5: (0)              import nltk
6: (0)              from textblob.base import CONTINUOUS, DISCRETE, BaseSentimentAnalyzer
7: (0)              from textblob.decorators import requires_nltk_corpus
8: (0)              from textblob.en import sentiment as pattern_sentiment
9: (0)              from textblob.tokenizers import word_tokenize
10: (0)             class PatternAnalyzer(BaseSentimentAnalyzer):
11: (4)                 """Sentiment analyzer that uses the same implementation as the
12: (4)                 pattern library. Returns results as a named tuple of the form:
13: (4)                 ``Sentiment(polarity, subjectivity, [assessments])``
14: (4)                 where [assessments] is a list of the assessed tokens and their
15: (4)                 polarity and subjectivity scores
16: (4)                 """
17: (4)                 kind = CONTINUOUS
18: (4)                 RETURN_TYPE = namedtuple("Sentiment", ["polarity", "subjectivity"])
19: (4)                 def analyze(self, text, keep_assessments=False):
20: (8)                     """Return the sentiment as a named tuple of the form:
21: (8)                     ``Sentiment(polarity, subjectivity, [assessments])``.
22: (8)                     """
23: (8)                     if keep_assessments:
24: (12)                        Sentiment = namedtuple(
25: (16)                            "Sentiment", ["polarity", "subjectivity", "assessments"]
26: (12)                        )
27: (12)                        assessments = pattern_sentiment(text).assessments
28: (12)                        polarity, subjectivity = pattern_sentiment(text)
29: (12)                        return Sentiment(polarity, subjectivity, assessments)
30: (8)                     else:
31: (12)                        Sentiment = namedtuple("Sentiment", ["polarity", "subjectivity"])
```

```
32: (12)                        return Sentiment(*pattern_sentiment(text))
33: (0)             def _default_feature_extractor(words):
34: (4)                 """Default feature extractor for the NaiveBayesAnalyzer."""
35: (4)                 return dict((word, True) for word in words)
36: (0)             class NaiveBayesAnalyzer(BaseSentimentAnalyzer):
37: (4)                 """Naive Bayes analyzer that is trained on a dataset of movie reviews.
38: (4)                 Returns results as a named tuple of the form:
39: (4)                 ``Sentiment(classification, p_pos, p_neg)``
40: (4)                 :param callable feature_extractor: Function that returns a dictionary of
41: (8)                     features, given a list of words.
42: (4)                 """
43: (4)                 kind = DISCRETE
44: (4)                 RETURN_TYPE = namedtuple("Sentiment", ["classification", "p_pos",
"p_neg"])
45: (4)                 def __init__(self, feature_extractor=_default_feature_extractor):
46: (8)                     super().__init__()
47: (8)                     self._classifier = None
48: (8)                     self.feature_extractor = feature_extractor
49: (4)                 @requires_nltk_corpus
50: (4)                 def train(self):
51: (8)                     """Train the Naive Bayes classifier on the movie review corpus."""
52: (8)                     super().train()
53: (8)                     neg_ids = nltk.corpus.movie_reviews.fileids("neg")
54: (8)                     pos_ids = nltk.corpus.movie_reviews.fileids("pos")
55: (8)                     neg_feats = [
56: (12)                        (
57: (16)
self.feature_extractor(nltk.corpus.movie_reviews.words(fileids=[f])),
58: (16)                            "neg",
59: (12)                        )
60: (12)                        for f in neg_ids
61: (8)                     ]
62: (8)                     pos_feats = [
63: (12)                        (
64: (16)
self.feature_extractor(nltk.corpus.movie_reviews.words(fileids=[f])),
65: (16)                            "pos",
66: (12)                        )
67: (12)                        for f in pos_ids
68: (8)                     ]
69: (8)                     train_data = neg_feats + pos_feats
70: (8)                     self._classifier =
nltk.classify.NaiveBayesClassifier.train(train_data)
71: (4)                 def analyze(self, text):
72: (8)                     """Return the sentiment as a named tuple of the form:
73: (8)                     ``Sentiment(classification, p_pos, p_neg)``
74: (8)                     """
75: (8)                     super().analyze(text)
76: (8)                     tokens = word_tokenize(text, include_punc=False)
77: (8)                     filtered = (t.lower() for t in tokens if len(t) >= 3)
78: (8)                     feats = self.feature_extractor(filtered)
79: (8)                     prob_dist = self._classifier.prob_classify(feats)
80: (8)                     return self.RETURN_TYPE(
81: (12)                        classification=prob_dist.max(),
82: (12)                        p_pos=prob_dist.prob("pos"),
83: (12)                        p_neg=prob_dist.prob("neg"),
84: (8)                     )
```

File 26 - SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRYCOMBINER_aligner_20_characters_for_pythons_codes.py:

```
1: (0)              import os
2: (0)              from datetime import datetime
3: (0)              def get_file_info(root_folder):
4: (4)                  file_info_list = []
5: (4)                  for root, dirs, files in os.walk(root_folder):
```

```
6: (8)                      for file in files:
7: (12)                         try:
8: (16)                             if file.endswith('.py'):
9: (20)                                 file_path = os.path.join(root, file)
10: (20)                                creation_time =
datetime.fromtimestamp(os.path.getctime(file_path))
11: (20)                                modified_time =
datetime.fromtimestamp(os.path.getmtime(file_path))
12: (20)                                file_extension = os.path.splitext(file)[1].lower()
13: (20)                                file_info_list.append([file, file_path, creation_time,
modified_time, file_extension, root])
14: (12)                        except Exception as e:
15: (16)                            print(f"Error processing file {file}: {e}")
16: (4)                 file_info_list.sort(key=lambda x: (x[2], x[3], len(x[0]), x[4]))  # Sort
by creation, modification time, name length, extension
17: (4)                 return file_info_list
18: (0)             def process_file(file_info_list):
19: (4)                 combined_output = []
20: (4)                 for idx, (file_name, file_path, creation_time, modified_time,
file_extension, root) in enumerate(file_info_list):
21: (8)                     with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
22: (12)                        content = f.read()
23: (12)                        content = "\n".join([line for line in content.split('\n') if
line.strip() and not line.strip().startswith("#")])
24: (12)                        content = content.replace('\t', '    ')
25: (12)                        processed_lines = []
26: (12)                        for i, line in enumerate(content.split('\n')):
27: (16)                            leading_spaces = len(line) - len(line.lstrip(' '))
28: (16)                            line_number_str = f"{i+1}: ({leading_spaces})"
29: (16)                            padding = ' ' * (20 - len(line_number_str))
30: (16)                            processed_line = f"{line_number_str}{padding}{line}"
31: (16)                            processed_lines.append(processed_line)
32: (12)                        content_with_line_numbers = "\n".join(processed_lines)
33: (12)                        combined_output.append(f"File {idx + 1} - {file_name}:\n")
34: (12)                        combined_output.append(content_with_line_numbers)
35: (12)                        combined_output.append("\n" + "-"*40 + "\n")
36: (4)                 return combined_output
37: (0)             root_folder_path = '.'  # Set this to the desired folder
38: (0)             file_info_list = get_file_info(root_folder_path)
39: (0)             combined_output = process_file(file_info_list)
40: (0)             output_file =
'SANJOYNATHQHENOMENOLOGYGEOMEETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt'
41: (0)             with open(output_file, 'w', encoding='utf-8') as logfile:
42: (4)                 logfile.write("\n".join(combined_output))
43: (0)             print(f"Processed file info logged to {output_file}")

----------------------------------------
```