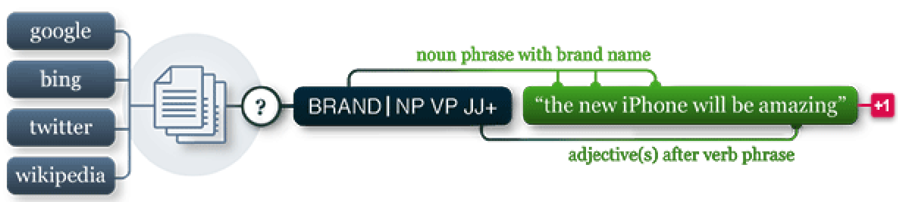
COMPUTATIONAL LINGUISTICS & PSYCHOLINGUISTICS RESEARCH CENTER

Search

pattern.en

The pattern.en module contains a fast part-of-speech tagger for English (identifies nouns, adjectives, verbs, etc. in a sentence), sentiment analysis, tools for English verb conjugation and noun singularization & pluralization, and a WordNet interface.

It can be used by itself or with other pattern (/pages/pattern) modules: web (/pages/pattern-web) | db (/pages/pattern-db) | en | search (/pages/pattern-search) | vector (/pages/pattern-vector) | graph (/pages/pattern-graph) .



Documentation

Indefinite article

The article is the most common determiner (**DT**) in English. It defines whether the successive noun is definite (*the cat*) or indefinite (*a cat*). The definite article is always *the*. The indefinite article can be *a* or *an* depending on how the successive noun is pronounced.

```
article(word, function=INDEFINITE)   # DEFINITE | INDEFINITE
```

```
referenced(word, article=INDEFINITE) # Returns article + word.
```

```
>>> from pattern.en import referenced
>>>
>>> print referenced('university')
>>> print referenced('hour')

a university
an hour
```

Reference: Granger, M. (2006). *Ruby Linguistics Framework*, http://deveiate.org/projects/Linguistics

Pluralization + singularization

The `pluralize()` function returns the singular form of a plural noun. The `singularize()` function returns the plural form of a singular noun. The pos parameter (part-of-speech) can be set to `NOUN` or `ADJECTIVE`, but only a small number of possessive adjectives inflect (e.g. *my* → *our*). The `custom` dictionary is for user-defined replacements. Accuracy is 96%.

```
pluralize(word, pos=NOUN, custom={}, classical=True)
```

```
singularize(word, pos=NOUN, custom={})
```

```
>>> from pattern.en import pluralize, singularize
>>>
>>> print pluralize('child')
>>> print singularize('wolves')

children
wolf
```

Reference:
Conway, D. (1998). An Algorithmic Approach to English Pluralization. *Proceedings of the 2nd Perl conference.*
Ferrer, B. (2005). *Inflector for Python*, http://www.bermi.org/projects/inflector

Comparative + superlative

The `comparative()` and `superlative()` functions give the comparative or superlative form of an adjective. Words with three or more syllables (e.g., *fantastic*) are simply preceded by *more* or *most*.

```
comparative(adjective)      # big => bigger
```

```
superlative(adjective)      # big => biggest
```

```
>>> from pattern.en import comparative, superlative
>>>
>>> print comparative('bad')
>>> print superlative('bad')

worse
worst
```

Verb conjugation

The pattern.en module has a lexicon of 8,500 common English verbs and their conjugated forms (infinitive, 3rd singular present, present participle, past and past participle – verbs such as *be* may have more forms). Some verbs can also be negated, including *be*, *can*, *do*, *will*, *must*, *have*, *may*, *need*, *dare*, *ought*.

```
conjugate(verb,
    tense = PRESENT,        # INFINITIVE, PRESENT, PAST, FUTURE
   person = 3,              # 1, 2, 3 or None
   number = SINGULAR,       # SG, PL
     mood = INDICATIVE,     # INDICATIVE, IMPERATIVE, CONDITIONAL, SUBJUNCTIVE
   aspect = IMPERFECTIVE,   # IMPERFECTIVE, PERFECTIVE, PROGRESSIVE
  negated = False,          # True or False
    parse = True)
```

```
lemma(verb)                 # Base form, e.g., are => be.
```

```
lexeme(verb)                # List of possible forms: be => is, was, ...
```

```
tenses(verb)                # List of possible tenses of the given form.
```

The `conjugate()` function takes the following optional parameters:

Tense	Person	Number	Mood	Aspect	Alias	Tag	Example
INFINITIVE	None	None	None	None	"inf"	**VB**	*be*
PRESENT	1	SG	INDICATIVE	IMPERFECTIVE	"1sg"	**VBP**	*I am*
PRESENT	2	SG	INDICATIVE	IMPERFECTIVE	"2sg"	·	*you are*
PRESENT	3	SG	INDICATIVE	IMPERFECTIVE	"3sg"	**VBZ**	*he is*
PRESENT	None	PL	INDICATIVE	IMPERFECTIVE	"pl"	·	*are*
PRESENT	None	None	INDICATIVE	PROGRESSIVE	"part"	**VBG**	*being*

PAST	None	None	None	None	"p"	**VBD**	*were*
PAST	1	PL	INDICATIVE	IMPERFECTIVE	"1sgp"	·	*I was*
PAST	2	PL	INDICATIVE	IMPERFECTIVE	"2sgp"	·	*you were*
PAST	3	PL	INDICATIVE	IMPERFECTIVE	"3gp"	·	*he was*

PAST	None	PL	INDICATIVE	IMPERFECTIVE	"ppl"	·	*were*
PAST	None	None	INDICATIVE	PROGRESSIVE	"ppart"	**VBN**	*been*

Instead of optional parameters, a single short alias, the part-of-speech tag, or `PARTICIPLE` or `PAST+PARTICIPLE` can also be given. With no parameters, the infinitive form of the verb is returned.

For example:

```
>>> from pattern.en import conjugate, lemma, lexeme
>>>
>>> print lexeme('purr')
>>> print lemma('purring')
>>> print conjugate('purred', '3sg') # he / she / it

['purr', 'purrs', 'purring', 'purred']
purr
purrs
```

```
>>> from pattern.en import tenses, PAST, PL
>>>
>>> print 'p' in tenses('purred') # By alias.
>>> print PAST in tenses('purred')
>>> print (PAST, 1, PL) in tenses('purred')

True
True
True
```

Reference: *XTAG English morphology* (1999), University of Pennsylvania, http://www.cis.upenn.edu/~xtag

RULE-BASED CONJUGATION

All verb functions have an optional `parse` parameter (`True` by default) that enables a rule-based parser for unknown verbs. This will not work for irregular verbs, and it is fragile for verbs ending in -e in the past tense, or the present participle. The overall accuracy of the algorithm is 91%.

With `parse=False`, `conjugate()` and `lemma()` yield `None`:

```
>>> from pattern.en import verbs, conjugate, PARTICIPLE
>>>
>>> print 'google'  in verbs.infinitives
>>> print 'googled' in verbs.inflections
>>>
>>> print conjugate('googled', tense=PARTICIPLE, parse=False)
>>> print conjugate('googled', tense=PARTICIPLE, parse=True)

False
False
None
googling
```

Quantification

The `number()` function returns a `float` or `int` parsed from the given (numeric) string. If no number can be parsed from the string, it returns `0`.

The `numerals()` function returns the given `int` or `float` as a string of numerals. By default, the fraction is rounded to two decimals.

The `quantify()` function returns a word count approximation. Two similar words are a *pair*, three to eight *several*, and so on. Words can be given as a list, a word → count dictionary, or as a single word + amount.

The `reflect()` function quantifies Python objects – see the examples bundled with the module.

```
number(string)              # "seventy-five point two" => 75.2
```

```
numerals(n, round=2)        # 2.245 => "two point twenty-five"
```

```
quantify([word1, word2, ...], plural={})
```

```
reflect(object, quantify=True, replace=[])
```

```
>>> from pattern.en import quantify
>>>
>>> print quantify(['goose', 'goose', 'duck', 'chicken', 'chicken', 'chicken'])
>>> print quantify({'carrot': 100, 'parrot': 20})
>>> print quantify('carrot', amount=1000)

several chickens, a pair of geese and a duck
dozens of carrots and a score of parrots
```

```
hundreds of carrots
```

Spelling

The `suggest()` function returns a list of spelling suggestions for a given word. Each suggestion is a `(word, confidence)`-tuple. It is about 70% accurate.

```
suggest(string)
```

```
>>> from pattern.en import suggest
>>> print suggest("parot")

[("part", 0.99), ("parrot", 0.01)]
```

Reference: Norvig, P. (2007). *How to Write a Spelling Corrector*. http://norvig.com/spell-correct.html

n-grams

The `ngrams()` function returns a list of *n*-grams (i.e., tuples of *n* successive words) from the given string. Alternatively, you can supply a `Text` or `Sentence` object (see further). Punctuation marks are stripped from words, and *n*-grams will not run over sentence delimiters (i.e., .!?), unless `continuous` is `True`.

```
ngrams(string, n=3, punctuation=".,;:!?()[]{}`''\"@#$^&*+-|=~_", continuous=False)
```

```
>>> from pattern.en import ngrams
>>> print ngrams("I am eating pizza.", n=2) # bigrams

[('I', 'am'), ('am', 'eating'), ('eating', 'pizza')]
```

Parser

A parser identifies sentences, words and word types in a string of text. This involves tokenization (distinguishing between abbreviations and sentence breaks), part-of-speech tagging (annotating words with their type, e.g., is *can* a **NOUN** or a **VERB**?) and chunking (grouping consecutive words that belong together). Parsing can be used to answer questions such as *who did what and why* and is useful in a wide range of text mining applications. The pattern.en parser uses a lexicon of a 100,000 known words and their part-of-speech tag (/pages/MBSP-tags), along with rules for unknown words based on word suffix (e.g., -*ly* = **ADVERB**) and context (surrounding words). This approach is fast but not always accurate, since many words are ambiguous and hard to capture with simple rules. The overall accuracy is about 95% (95.8% on WSJ portions 22-24). It is lower for informal language use (e.g., chat language).

The `parse()` function takes a string of text and returns a part-of-speech tagged Unicode string. Sentences in the output are separated by newline characters.

```
parse(string,
   tokenize = True,         # Split punctuation marks from words?
       tags = True,         # Parse part-of-speech tags? (NN, JJ, ...)
     chunks = True,         # Parse chunks? (NP, VP, PNP, ...)
  relations = False,        # Parse chunk relations? (-SBJ, -OBJ, ...)
    lemmata = False,        # Parse lemmata? (ate => eat)
   encoding = 'utf-8'       # Input string encoding.
     tagset = None)         # Penn Treebank II (default) or UNIVERSAL.
```

For example:

```
>>> from pattern.en import parse
>>> print parse('I eat pizza with a fork.')

I/PRP/B-NP/O eat/VBD/B-VP/O pizza/NN/B-NP/O with/IN/B-PP/B-PNP a/DT/B-NP/I-PNP
fork/NN/I-NP/I-PNP ././O/O
```

- With `tags=True` each word is annotated with a part-of-speech tag.
- With `chunks=True` each word is annotated with a chunk tag and a **PNP** tag (prepositional noun phrase, **PP** + **NP**). The **O** tag (= outside) means that the word is not part of a chunk.
- With `relations=True` each word is annotated with a role tag (e.g., **-SBJ** for subject or -**OBJ** for).
- With `lemmata=True` each word is annotated with its base form.
- With `tokenize=False`, punctuation marks will not be separated from words.
  The input string is expected to be tokenized beforehand, or sentence delimiters are not discovered.

Reference: Brill, E. (1992). *A simple rule-based part of speech tagger*. ANLC '92 Proceedings.

Parser tags

Let's examine the word *fork* and the tags assigned by the parser in the example above:

WORD	PART-OF-SPEECH	CHUNK	PNP
fork	**NN**	**I-NP**	**I-PNP**

The word's part-of-speech tag is **NN**, which means that it is a noun. The word occurs in a **NP** chunk, a noun phrase (i.e., *a fork*). It is also part of a prepositional noun phrase (i.e., *with a fork*).

Common part-of-speech tags are **NN** (noun), **VB** (verb), **JJ** (adjective), **RB** (adverb) and **IN** (preposition).
Common chunk tags are **NP** (noun phrase) and **VP** (verb phrase).
Common chunk relations are **NP-SBJ** (subject) and **NP-OBJ** (object).

The Penn Treebank II tagset (/pages/MBSP-tags) gives an overview of all the possible tags generated by the parser.

Parser tagger & tokenizer

The `tokenize()` function returns a list of sentences, with punctuation marks split from words. It takes an optional `replace` dictionary, by default used to split contractions, i.e., `{"'ve": " 've", ...}`.

The `tag()` function simply annotates words with their part-of-speech tag and returns a list of `(word, tag)`-tuples:

```
tokenize(string, punctuation=".,;:!?()[]{}`''\"@#$^&*+-|=~_", replace={})
```

```
tag(string, tokenize=True, encoding='utf-8')
```

```
>>> from pattern.en import tag
>>>
>>> for word, pos in tag('I feel *happy*!')
>>>     if pos == "JJ": # Retrieve all adjectives.
>>>         print word

happy
```

Parser output

The output of `parse()` is a string of sentences in which each word has been annotated with the requested tags. The `pprint()` function gives a human-readable breakdown of the tags (the extra *p-* is for *pretty*).

```
>>> from pattern.en import parse
>>> from pattern.en import pprint
>>>
>>> pprint(parse('I ate pizza.', relations=True, lemmata=True))

    WORD   TAG    CHUNK   ROLE   ID     PNP    LEMMA
       I   PRP    NP      SBJ    1      -      i
     ate   VBP    VP      -      1      -      eat
   pizza   NN     NP      OBJ    1      -      pizza
       .   .      -       -      -      -      .
```

The output of `parse()` is a subclass of `unicode` called `TaggedString` whose `TaggedString.split()` method by default yields a list of sentences, where each sentence is a list of tokens, where each token is a list of the word + its tags.

```
>>> from pattern.en import parse
>>> print parse('I ate pizza.').split()

[[[u'I', u'PRP', u'B-NP', u'O'],
  [u'ate', u'VBD', u'B-VP', u'O'],
  [u'pizza', u'NN', u'B-NP', u'O'],
  [u'.', u'.', u'O', u'O']]]
```

The most convenient way to analyze and mine the output is to construct a parse tree (#tree).

Parse trees

A parse tree stores a tagged string as a tree of nested objects that can be traversed to analyze the constituents in the text. The `parsetree()` function takes the same parameters as `parse()` and returns a `Text` object. A `Text` is a list of `Sentence` objects. Each `Sentence` is a list of `Word` objects. `Word` objects can be grouped in `Chunk` objects, which are related to other `Chunk` objects.

```
parsetree(string,
   tokenize = True,         # Split punctuation marks from words?
       tags = True,         # Parse part-of-speech tags? (NN, JJ, ...)
     chunks = True,         # Parse chunks? (NP, VP, PNP, ...)
  relations = False,        # Parse chunk relations? (-SBJ, -OBJ, ...)
    lemmata = False,        # Parse lemmata? (ate => eat)
   encoding = 'utf-8'       # Input string encoding.
     tagset = None)         # Penn Treebank II (default) or UNIVERSAL.
```

The following example shows the parse tree for the sentence "*The cat sat on the mat.*":

```
>>> from pattern.en import parsetree
```

```
>>>
>>> s = parsetree('The cat sat on the mat.', relations=True, lemmata=True)
>>> print repr(s)

[Sentence(
 u'The/DT/B-NP/O/NP-SBJ-1/the
   cat/NN/I-NP/O/NP-SBJ-1/cat
   sat/VBD/B-VP/O/VP-1/sit
   on/IN/B-PP/B-PNP/O/on
   the/DT/B-NP/I-PNP/O/the
   mat/NN/I-NP/I-PNP/O/mat
   ././O/O/O/O/.')]
```

```
>>> for sentence in s:
>>>     for chunk in sentence.chunks:
>>>         print chunk.type, [(w.string, w.type) for w in chunk.words]

NP [(u'the', u'DT'), (u'cat', u'NN')]
VP [(u'sat', u'VBD')]
PP [(u'on', u'IN')]
NP [(u'the', 'DT), (u'mat', u'NN')]
```

A common approach is to store output from `parse()` in a .txt file, with a tagged sentence on each line. The `tree()` function can be used to load it as a `Text` object. It has an optional `token` parameter that defines the format of the tokens (tagged words). So `parsetree(s)` is the same as `tree(parse(s))`.

```
tree(taggedstring, token=[WORD, POS, CHUNK, PNP, REL, LEMMA])
```

```
>>> from pattern.en import tree
>>>
>>> for sentence in tree(open('tagged.txt'), token=[WORD, POS, CHUNK])
>>>     print sentence
```

Text

A `Text` is a list of `Sentence` objects (i.e., it can be iterated with `for sentence in text:`).

```
text = Text(taggedstring, token=[WORD, POS, CHUNK, PNP, REL, LEMMA])
```

```
text = Text.from_xml(xml)  # Reads an XML string generated with Text.xml.
```

```
text.string                # 'The cat sat on the mat .'
text.sentences             # [Sentence('The cat sat on the mat .')]
text.copy()
text.xml
```

Sentence

A `Sentence` is a list of `Word` objects, with attributes and methods that group words in `Chunk` objects.

```
sentence = Sentence(taggedstring, token=[WORD, POS, CHUNK, PNP, REL, LEMMA])
```

```
sentence = Sentence.from_xml(xml)
```

```
sentence.parent            # Sentence parent, or None.
sentence.id                # Unique id for each sentence.
sentence.start             # 0
sentence.stop              # len(Sentence).
```

```
sentence.string            # Tokenized string, without tags.
sentence.words             # List of Word objects.
sentence.lemmata           # List of word lemmata.
sentence.chunks            # List of Chunk objects.
sentence.subjects          # List of NP-SBJ chunks.
sentence.objects           # List of NP-OBJ chunks.
sentence.verbs             # List of VP chunks.
sentence.relations         # {'SBJ': {1: Chunk('the cat/NP-SBJ-1')},
                           #    'VP': {1: Chunk('sat/VP-1')},
                           #   'OBJ': {}}
sentence.pnp               # List of PNPChunks: [Chunk('on the mat/PNP')]
```

```
sentence.constituents(pnp=False)
```

```
sentence.slice(start, stop)
sentence.copy()
sentence.xml
```

- `Sentence.constituents()` returns a mixed, in-order list of `Word` and `Chunk` objects.
  With `pnp=True`, it will yield `PNPChunk` objects whenever possible.
- `Sentence.slice()` returns a `Slice` (= a subclass of `Sentence`) starting with the word at index `start` and containing all words up to (not including) index `stop`.

Sentence words

A `Sentence` is made up of `Word` objects, which are also grouped in `Chunk` objects:

```
word = Word(sentence, string, lemma=None, type=None, index=0)
```

```
word.sentence              # Sentence parent.
word.index                 # Sentence index of word.
word.string                # String (Unicode).
word.lemma                 # String lemma, e.g. 'sat' => 'sit',
word.type                  # Part-of-speech tag (NN, JJ, VBD, ...)
word.chunk                 # Chunk parent, or None.
word.pnp                   # PNPChunk parent, or None.
```

Sentence chunks

A `Chunk` is a list of `Word` objects that belong together.
Multiple chunks can be part of a `PNPChunk`, which start with a **`PP`** chunk followed by **`NP`** chunks.

```
chunk = Chunk(sentence, words=[], type=None, role=None, relation=None)
```

```
chunk.sentence             # Sentence parent.
chunk.start                # Sentence index of first word.
chunk.stop                 # Sentence index of last word + 1.
chunk.string               # String of words (Unicode).
chunk.words                # List of Word objects.
chunk.lemmata              # List of word lemmata.
chunk.head                 # Primary Word in the chunk.
chunk.type                 # Chunk tag (NP, VP, PP, ...)
chunk.role                 # Role tag (SBJ, OBJ, ...)
chunk.relation             # Relation id, e.g. NP-SBJ-1 => 1.
chunk.relations            # List of (id, role)-tuples.
chunk.related              # List of Chunks with same relation id.
chunk.subject              # NP-SBJ chunk with same id.
chunk.object               # NP-OBJ chunk with same id.
chunk.verb                 # VP chunk with same id.
chunk.modifiers            # []
chunk.conjunctions         # []
chunk.pnp                  # PNPChunk parent, or None.
```

```
chunk.previous(type=None)
chunk.next(type=None)
chunk.nearest(type='VP')
```

- `Chunk.head` yields the primary `Word` in the chunk: *the big cat* → *cat*.
- `Chunk.relations` contains all relations the chunk is part of.
  Some chunks have multiple relations, e.g., **`SBJ`** as well as **`OBJ`**, or **`OBJ`** of multiple **`VP`**'s.
- For **`VP`** chunks, `Chunk.modifiers` is a list of nearby adjectives and adverbs that have no relations.
  For example, in *the cat purred happily*, modifier of *purred* → *happily*.
- `Chunk.conjunctions` is a list of chunks linked by *and* and *or* to this chunk.
  For example in *up and down*: the *up* chunk has conjunctions: `[(Chunk('down'), AND)]`.

Prepositional noun phrases

A `PNPChunk` or prepositional noun phrase is a subclass of `Chunk`. It groups **`PP`** + **`NP`** chunks (= **`PNP`**).

```
pnp = PNPChunk(sentence, words=[], type=None, role=None, relation=None)
```

```
pnp.string                 # String of words (Unicode).
pnp.chunks                 # List of Chunk objects.
pnp.preposition            # First PP chunk in the PNP.
```

Words and chunks that are part of a **`PNP`** will have their `Word.pnp` and `Chunk.pnp` attribute set. All prepositional noun phrases in a sentence can be retrieved with `Sentence.pnp`.

Sentiment

Written text can be broadly categorized into two types: facts and opinions. Opinions carry people's sentiments, appraisals and feelings toward the world. The pattern.en module bundles a lexicon of adjectives (e.g., *good*, *bad*, *amazing*, *irritating*, ...) that occur frequently in product reviews, annotated with scores for sentiment polarity (positive ↔ negative) and subjectivity (objective ↔ subjective).

The `sentiment()` function returns a `(polarity, subjectivity)`-tuple for the given sentence, based on the adjectives it contains, where polarity is a value between `-1.0` and `+1.0` and subjectivity between `0.0` and `1.0`. The sentence can be a string, `Text`, `Sentence`, `Chunk`, `Word` or a `Synset` (see below).

The `positive()` function returns `True` if the given sentence's polarity is above the threshold. The threshold can be lowered or raised, but overall `+0.1` gives the best results for product reviews. Accuracy is about 75% for movie reviews.

```
sentiment(sentence)        # Returns a (polarity, subjectivity)-tuple.
```

```
positive(s, threshold=0.1) # Returns True if polarity >= threshold.
```

```
>>> from pattern.en import sentiment
>>>
>>> print sentiment(
>>>     "The movie attempts to be surreal by incorporating various time paradoxes,"
>>>     "but it's presented in such a ridiculous way it's seriously boring.")

(-0.34, 1.0)
```

In the example above, `-0.34` is the average of *surreal*, *various*, *ridiculous* and *seriously boring*. To retrieve the scores for individual words, use the special `assessments` property, which yields a list of `(words, polarity, subjectivity, label)`-tuples.

```
>>> print sentiment('Wonderfully awful! :-)').assessments

[(['wonderfully', 'awful', '!'], -1.0, 1.0, None),
 ([':-)'], 0.5, 1.0, 'mood')]
```

Mood & modality

Grammatical mood refers to the use of auxiliary verbs (e.g., *could*, *would*) and adverbs (e.g., *definitely*, *maybe*) to express uncertainty.

The `mood()` function returns either `INDICATIVE`, `IMPERATIVE`, `CONDITIONAL` or `SUBJUNCTIVE` for a given parsed `Sentence`. See the table below for an overview of moods.

The `modality()` function returns the degree of certainty as a value between `-1.0` and `+1.0`, where values > `+0.5` represent facts. For example, "*I wish it would stop raining*" scores `-0.35`, whereas "*It will stop raining*" scores `+0.75`. Accuracy is about 68% for Wikipedia texts.

```
mood(sentence)     # Returns INDICATIVE | IMPERATIVE | CONDITIONAL | SUBJUNCTIVE
```

```
modality(sentence) # Returns -1.0 => +1.0.
```

Mood	Form	Use	Example
`INDICATIVE`	none of the below	fact, belief	*It rains.*
`IMPERATIVE`	infinitive without *to*	command, warning	*Don't rain!*
`CONDITIONAL`	*would*, *could*, *should*, *may*, or *will*, *can* + *if*	conjecture	*It might rain.*
`SUBJUNCTIVE`	*wish*, *were*, or *it is* + infinitive	wish, opinion	*I hope it rains.*

For example:

```
>>> from pattern.en import parse, Sentence, parse
>>> from pattern.en import modality
>>>
>>> s = "Some amino acids tend to be acidic while others may be basic." # weaseling
>>> s = parse(s, lemmata=True)
>>> s = Sentence(s)
>>>
>>> print modality(s)

0.11
```

WordNet

The pattern.en.wordnet module includes WordNet 3.0 and Oliver Steele's PyWordNet module. [WordNet (http://wordnet.princeton.edu/)](http://wordnet.princeton.edu/) is a lexical database that groups related words into `Synset` objects (= sets of synonyms). Each synset provides a short definition and semantic relations to other synsets.

The `synsets()` function returns a list of `Synset` objects for a given word, where each set corresponds to a word sense (e.g., *tree* in the sense of plant, *tree* in the sense of diagram, etc.)

```
synset = wordnet.synsets(word, pos=NOUN)[i]
```

```
synset.pos                  # Part-of-speech: NOUN | VERB | ADJECTIVE | ADVERB.
synset.synonyms             # List of word forms (i.e., synonyms).
synset.gloss                # Definition string.
synset.lexname              # Category string, or None.
synset.ic                   # Information Content (float).
```

```
synset.antonym              # Synset (semantic opposite).
synset.hypernym             # Synset (semantic parent).
```

```
synset.hypernyms(recursive=False, depth=None)
synset.hyponyms(recursive=False, depth=None)
synset.meronyms()          # List of synsets (members/parts).
synset.holonyms()          # List of synsets (of which this is a member).
synset.similar()           # List of synsets (similar adjectives/verbs).
```

- `Synset.hypernyms()` returns a list of parent synsets (i.e., more general).
- `Synset.hyponyms()` returns a list child synsets (i.e., more specific).
  With `recursive=True`, returns parents of parents or children of children.
  Optionally, returns parents or children recursively up to the given `depth`.

For example:

```
>>> from pattern.en import wordnet
>>>
>>> s = wordnet.synsets('bird')[0]
>>>
>>> print 'Definition:', s.gloss
>>> print '  Synonyms:', s.synonyms
>>> print ' Hypernyms:', s.hypernyms()
>>> print '  Hyponyms:', s.hyponyms()
>>> print '  Holonyms:', s.holonyms()
>>> print '  Meronyms:', s.meronyms()

Definition: u'warm-blooded egg-laying vertebrates characterized '
             'by feathers and forelimbs modified as wings'
  Synonyms: [u'bird']
 Hypernyms: [Synset(u'vertebrate')]
  Hyponyms: [Synset(u'cock'), Synset(u'hen'), ...]
  Holonyms: [Synset(u'Aves'), Synset(u'flock')]
  Meronyms: [Synset(u'beak'), Synset(u'feather'), ...]
```

Reference: Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*. Cambridge, MIT Press.

Synset similarity

The `ancestor()` function returns the common ancestor of two synsets. The `similarity()` function returns the semantic similarity of two synsets as a value between `0.0`–`1.0`.

```
wordnet.ancestor(synset1, synset2)
```

```
wordnet.similarity(synset1, synset2)
```

```
>>> from pattern.en import wordnet
>>>
>>> a = wordnet.synsets('cat')[0]
>>> b = wordnet.synsets('dog')[0]
>>> c = wordnet.synsets('box')[0]
>>>
>>> print wordnet.ancestor(a, b)
>>>
>>> print wordnet.similarity(a, a)
>>> print wordnet.similarity(a, b)
>>> print wordnet.similarity(a, c)

Synset('carnivore')
1.0
0.86
0.17
```

Similarity is calculated using Lin's formula and Resnik's Information Content (IC). IC values for each synset are derived from the word count in Brown corpus.

lin = `2.0` * `log(ancestor(synset1, synset2).ic)` / `log(synset1.ic` * `synset2.ic)`

Synset sentiment

SentiWordNet (http://sentiwordnet.isti.cnr.it/) is a lexical resource for opinion mining, with polarity and subjectivity scores for all WordNet synsets. SentiWordNet is free for non-commercial research purposes. To use SentiWordNet, request a download from the authors and put `SentiWordNet*.txt` in `pattern/en/wordnet/`. You can then use `Synset.weight()` in your script:

```
>>> from pattern.en import wordnet
>>> from pattern.en import ADJECTIVE
>>>
>>> print wordnet.synsets('happy', ADJECTIVE)[0].weight
>>> print wordnet.synsets('sad', ADJECTIVE)[0].weight

(0.375, 0.875)
(-0.625, 0.875)
```

Wordlists

The patten.en module includes a number of general-purpose word lists:

List	Description	Size	Example
ACADEMIC	English academic words	500	*criterion, proportionally, research*
BASIC	English basic words	1,000	*chicken, pain, road*
PROFANITY	English swear words	350	
TIME	English time & date words	100	*Christmas, past, saturday*

```
>>> from pattern.en.wordlist import ACADEMIC
>>>
>>> words = open('paper.txt').read().split()
>>> words = [w for w in words if w not in ACADEMIC]
```

See also

- [MBSP (/pages/MBSP)](/pages/MBSP) (GPL): robust parser using a memory-based learning approach, in Python.
- [NLTK (http://www.nltk.org/)](http://www.nltk.org/) (Apache): full natural language processing toolkit for Python.