The NLTK FrameNet API: Designing for Discoverability with a Rich Linguistic Resource

Nathan Schneider
Georgetown University
Washington, DC
nathan.schneider@georgetown.edu

Chuck Wooters
Semantic Machines
Berkeley, CA
wooters@semanticmachines.com

Abstract
A new Python API, integrated within the NLTK suite, offers access to the FrameNet 1.7 lexical database. The lexicon (structured in terms of frames) as well as annotated sentences can be processed programmatically, or browsed with human-readable displays via the interactive Python prompt.

1 Introduction
For over a decade, the Berkeley FrameNet (henceforth, simply “FrameNet”) project (Baker et al., 1998) has been documenting the vocabulary of contemporary English with respect to the theory of frame semantics (Fillmore, 1982). A freely available, linguistically-rich resource, FrameNet now covers over 1,000 semantic frames, 10,000 lexical senses, and 100,000 lexical annotations in sentences drawn from corpora. The resource has formed a basis for much research in natural language processing—most notably, a tradition of semantic role labeling that continues to this day (Gildea and Jurafsky, 2002; Baker et al., 2007; Das et al., 2014; FitzGerald et al., 2015; Roth and Lapata, 2015, inter alia).

Despite the importance of FrameNet, computational users are often frustrated by the complexity of its custom XML format. Whereas much of the resource is browsable on the web (http://framenet.icsi.berkeley.edu/), certain details of the linguistic descriptions and annotations languish in obscurity as they are not exposed by the HTML views of the data.1 The few open source APIs for reading FrameNet data are now antiquated, and none has been widely adopted.2

We describe a new, user-friendly Python API for accessing FrameNet data. The API is included within recent releases of the popular NLTK suite (Bird et al., 2009), and provides access to nearly all the information in the FrameNet release.

2 Installation
Instructions for installing NLTK are found at nltk.org. NLTK is cross-platform and supports Python 2.7 as well as Python 3.x environments. It is bundled in the Anaconda and Enthought Canopy Python distributions for data scientists.3

In a working NLTK installation (version 3.2.2 or later), one merely has to invoke a method to download the FrameNet data:4,5

>>> import nltk
>>> nltk.download('framenet_v17')
Subsequently, the framenet module is loaded as follows (with alias fn for convenience):

```python
>>> from nltk.corpus import framenet as fn
```

## 3 Overview of FrameNet

FrameNet is organized around conceptual structures known as frames. A semantic frame represents a scene—a kind of event, state, or other scenario which may be universal or culturally-specific, and domain-general or domain-specific. The frame defines participant roles or frame elements (FEs), whose relationships forms the conceptual background required to understand (certain senses of) vocabulary items. Oft-cited examples by Fillmore include:

- **Verbs** such as *buy, sell*, and *pay*, and nouns such as *buyer, seller, price*, and *purchase*, are all defined with respect to a commercial transaction scene (frame). FEs that are central to this frame—they may or may not be mentioned explicitly in a text with one of the aforementioned lexical items—are the *Buyer, the Seller, the Goods* being sold by the *Seller, and the Money* given as payment in exchange by the *Buyer*.

- The concept of REVENGE—lexicalized in vocabulary items such as *revenge, avenge, avenger, retaliate, payback*, and get even—fundamentally presupposes an Injury that an Offender has inflicted upon an Injured_party for which an Avenger (who may or may not be the same as the Injured_party) seeks to exact some Punishment on the Offender.

- A hypotenuse presupposes a geometrical notion of right triangle, while a pedestrian presupposes a street with both vehicular and nonvehicular traffic. (Neither frame is currently present in FrameNet.)

The FEs in a frame are formally listed alongside an English description of their function within the frame. Frames are organized in a network, including an inheritance hierarchy (e.g., REVENGE is a special case of an EVENT) and other kinds of frame-to-frame relations.

Vocabulary items listed within a frame are called **lexical units** (LUs). FrameNet’s inventory of LUs includes both content and function words. Formally, an LU links a lemma with a frame.\(^6\)

\(^6\)The lemma name incorporates a part-of-speech tag. The lemma may consist of a single word, such as *surrender.v*, or multiple words, such as *give up.v*.

In a text, a token of an LU is said to **evoke** the frame. Sentences are annotated with respect to frame-evoking tokens and their FE spans. Thus:

```
[Snape]Injured_party ’s revenge [on Harry]Offender
```

The reader is referred to (Fillmore and Baker, 2009) for a contemporary introduction to the resource and the theory of frame semantics upon which it is based. Extensive linguistic details are provided in (Ruppenhofer et al., 2016).

## 4 API Overview

### 4.1 Design Principles

The API is designed with the following goals in mind:

**Simplicity.** It should be easy to access important objects in the database (primarily frames, lexical units, and annotations), whether by iterating over all entries or searching for particular ones. To avoid cluttering the API with too many methods, other information in the database should be reachable via object attributes. Calling the API’s `help()` method prints a summary of the main methods for accessing information in the database.

**Discoverability.** Many of the details of the database are complex. The API makes it easy to browse what is in database objects via the Python interactive prompt. The main way it achieves this is with pretty-printed displays of the objects, such as the frame display in figure 1 (see §4.3). The display makes it clear how to access attributes of the object that a novice user of FrameNet might not have known about.

In our view, this approach sets this API apart from others. Some of the other NLTK APIs for complex structured data make it difficult to browse the structure without consulting documentation.

**On-demand loading.** The database is stored in thousands of XML files, including files indexing the lists of frames, frame relations, LUs, and full-text documents, plus individual files for all frames, LUs, and full-text documents. Unzipped, the FrameNet 1.7 release is 855 MB. Loading all of these files—particularly the corpus annotations—is slow and memory-intensive, costs which are unnecessary for many purposes. Therefore, the API is carefully designed with lazy data structures to load XML files only as needed. Once loaded, all data is cached in memory for fast subsequent access.
Frame (347): Revenge

[URL] https://framenet2.icsi.berkeley.edu/fnReports/data/frame/Revenge.xml

[Definition]
This frame concerns the infliction of punishment in return for a
wrong suffered. An Avenger performs a Punishment on an Offender
as a consequence of an earlier action by the Offender, the Injury.
The Avenger inflicting the Punishment need not be the same as the
Injured Party who suffered the Injury, but the Avenger does have
to share the judgment that the Offender's action was wrong. The
judgment that the Offender had inflicted an Injury is made
without regard to the law. '(1) They took revenge for the deaths
of two loyalist prisoners.' '(2) Lachlan went out to avenge
them.' '(3) The next day, the Roman forces took revenge on their
enemies.'

[SemTypes] 0 semantic types

[FrameRelations] 1 frame relations
  <Parent=Rewards_and_punishments -- Inheritance -> Child=Revenge>

[LexUnit] 18 lexical units
  avenge.v (6056), avenger.n (6057), get back (at).v (10003), get
even.v (6075), payback.n (10124), retaliate.v (6065),
  retaliation.n (6071), retribution.n (6070), retributive.a (6074),
  retaliatory.a (6076), revenge.n (6067), revenge.v (6066),
  revengeful.a (6073), reverter.n (6072), sanction.n (10676),
  vengeance.n (6058), vengeful.a (6068), vindictive.a (6069)

[FE] 14 frame elements
  Core: Avenger (3009), Injured_party (3022), Injury (3018),
  Offender (3012), Punishment (3015)
  Peripheral: Degree (3010), Duration (12060), Instrument (3013),
  Manner (3014), Place (3016), Purpose (3017), Time (3021)
  Extra-Thematic: Depictive (3011), Result (3020)

[FECoreSets] 2 frame element core sets
  Injury, Injured_party
  Avenger, Punishment

Figure 1: Textual display of the Revenge frame. Shown in square brackets are attribute names for accessing the frame’s contents. In parentheses are IDs for the frame, its LUs, and its FEs.

4.2 Lexicon Access Methods

The main methods for looking up information in
the lexicon are:

frames(name) frame(nameOrId)
lus(name, frame) lu(id)
fes(name, frame)

The methods with plural names (left) are for searching
the lexicon by regular expression pattern to be
matched against the entry name. In addition (or
instead), lus() and fes() allow for the results to
be restricted to a particular frame. The result is a
list with 0 or more elements. If no arguments are
provided, all entries in the lexicon are returned.

An example of a search by frame name pattern: 7

>>> fn.frames('(?i)creat')

[<frame ID=268 name=Cooking_creation>,
 <frame ID=1658 name=Create_physical_artwork>, ...]

Similarly, a search by LU name pattern—note
that the .v suffix is used for all verbal LUs:

>>> fn.lus(r'.+en\v')
[<lu ID=5331 name=awaken.v>,
 <lu ID=7544 name=betoken.v>, ...]

The frame() and lu() methods are for retrieving
a single known entry by its name or ID. At-
ttempting to retrieve a nonexistent entry triggers an
exception of type FramenetError.

Two additional methods are available for
frame lookup: frame_ids_and_names(name) to
get a mapping from frame IDs to names, and
frames_by_lemma(name) to get all frames with
some LU matching the given name pattern.

7 (?i) makes the pattern case-insensitive.
A short while later Joseph had his revenge on Watney’s.

---

Time Avenge sup Ave Offender

4.3 Database Objects

All structured objects in the database—frames, LUs, FEs, etc.—are loaded as AttrDict data structures. Each AttrDict instance is a mapping from string keys to values, which can be strings, numbers, or structured objects. AttrDict is so called because it allows keys to be accessed as attributes:

```python
>>> f = fn.frame('Revenge')
>>> f.keys()
dict_keys(['cBy', 'cDate', 'name', 'ID', '_type', 'definition', 'definitionMarkup', 'frameRelations', 'FE', 'FECoreSets', 'lexUnit', 'semTypes', 'URL'])
```

```python
>>> f.name
'Revenge'
```

```python
>>> f.ID
347
```

For the most important kinds of structured objects, the API specifies textual displays that organize the object’s contents in a human-readable fashion. Figure 1 shows the display for the REVENGE frame, which would be printed by entering `fn.frame('Revenge')` at the interactive prompt. The display gives attribute names in square brackets; e.g., `lexUnit`, which is a mapping from LU names to objects. Thus, after the code listing in the previous paragraph, `f.lexUnit['revenge.n']` would access to one of the LU objects in the frame, which in turn has its own attributes and textual display.

4.4 Advanced Lexicon Access

Frame relations. The inventory of frames is organized in a semantic network via several kinds of frame-to-frame relations. For instance, the REVENGE frame is involved in one frame-to-frame relation: it is related to the more general REWARDS_AND_PUNISHMENTS frame by Inheritance, as shown in the middle of figure 1. REWARDS_AND_PUNISHMENTS, in turn, is involved in Inheritance relations with other frames. Each frame-to-frame relation bundles mappings between corresponding FEs in the two frames.

Apart from the frameRelations attribute of frame objects, frame-to-frame relations can be browsed by the main method `frame_relations(frame, frame2, type)`, where the optional arguments allow for filtering by one or both frames and the kind of relation. Within a frame relation object, pairwise FE relations are stored in the feRelations attribute. Main method `fe_relations()` provides direct access to links between FEs. The inventory of relation types, including Inheritance, Causative, Inchoative, Subframe, Perspective_on, and others, is available.

```python
>>> f.name
'Revenge'
>>> f.ID
347
```
They’ve been looking for him all the time for their revenge, but it is only now that they have begun to find him out.

Figure 3: A sentence of full-text annotation. If this sentence object is stored under the variable `sent`, its frame annotation with respect to the target `revenge` is accessed as `sent.annotationSet[2]`. (The ? under `looking` indicates that there is no corresponding LU defined in the `Seeking` frame; in some cases the full-text annotators marked but did not define out-of-vocabulary LUs which fit an existing frame. Also, some full-text annotation sets annotate an LU without its FEs—these are shown with ! to reflect the annotation set’s status code of UNANN.)

via main method `frame_relation_types()`.

**Semantic types.** These provide additional semantic categorizations of FEs, frames, and LUs. For FEs, they mark selectional restrictions (e.g., `f.FE['Avenger'].semType` gives the `Sentient` type). Main method `propagate_semtypes()` propagates the FE semantic type labels marked explicitly to other FEs according to inference rules that follow the FE relations. This should be called prior to inspecting FE semtypes (it is not called by default because it takes several seconds to run).

The semantic types are database objects in their own right, and they are organized in their own inheritance hierarchy. Main method `semtypes()` returns all semantic types as a list; main method `semytype()` looks up a particular one by name, ID, or abbreviation; and main method `semtypes_inherits()` checks whether two semantic types have a subtype-supertype relationship.

4.5 Corpus Access

Frame-semantic annotations of sentences can be accessed via the `exemplars` and `subCorpus` attributes of an LU object, or via the following main methods:

```
annotions(luname, exemplars, full_text)
sents() exemplars(luname) ft_sents(docname)
doc(id) docs(name)
```

`annotions()` returns a list of frame annotation sets. Each annotation set consists of a frame-evoking target (token) within a sentence, the LU in the frame it evokes, its overt FE spans in the sentence, and the status of null-instantiated FEs. Optionally, the user may filter by LU name, or limit by the type of annotation (see next paragraph): `exemplars` and `full_text` both default to True. In the XML, the components of an annotation set are stored in several annotation layers: one (and sometimes more than one) layer of FEs, as well as additional layers for other syntactic information (including grammatical function and phrase type labels for each FE, and copular or support words relative to the frame-evoking target).

Annotation sets are organized by sentence. Corpus sentences appear in two kinds of annotation: `exemplars()` retrieves sentences with lexicographic annotation (where a single target has been selected for annotation to serve as an example of an LU); the optional argument allows for filtering the set of LUs. `ft_sents()` retrieves sentences from documents selected for full-text annotation (as many targets in the document as possible have been annotated); the optional argument allows for filtering by document name. `sents()` can be used to iterate over all sentences. Technically, each sentence object contains multiple annotation sets: the first is for sentence-level annotations, including the part-of-speech tagging and in some cases named entity labels; subsequent annotation sets are for

---

8. In frame semantics, core FEs that are not overt but are conceptually required by a frame are said to be implicit via null instantiation (Fillmore and Baker, 2009).
frame annotations. As lexicographic annotations have only one frame annotation set, it is visualized in the sentence display: figure 2 shows the display for f.lexUnit[‘revenge.n’].exemplars[20]. Full-text annotations display target information only, allowing the user to drill down to see each annotation set, as in figure 3.

Sentences of full-text annotation can also be browsed by document using the doc() and docs() methods. The document display lists the sentences with numeric offsets.

5 Limitations and future work

The main part of the Berkeley FrameNet data that the API currently does not support are valence patterns. For a given LU, the valence patterns summarize the FEs’ syntactic realizations across annotated tokens. They are displayed in each LU’s “Lexical Entry” report on the FrameNet website.

We intend to add support for valence patterns in future releases, along with more sophisticated querying/browsing capabilities for annotations, and better displays for syntactic information associated with FE annotations. Some of this functionality can be modeled after tools like FrameSQL (Sato, 2003) and Valencer (Kabbach and Ribeyre, 2016). In addition, it is worth investigating whether the API can be adapted for FrameNets in other languages, and to support cross-lingual mappings being added to 14 of these other FrameNets in the ongoing Multilingual FrameNet project.

Acknowledgments

We thank Collin Baker, Michael Ellsworth, and Miriam R. L. Petrick for helping us to understand the FrameNet annotation process and the technical aspects of the data, and for co-organizing the FrameNet tutorial in which an early version of the API was introduced (Baker et al., 2015). We also thank NLTK project leader Steven Bird, Mikhail Kbororov, Pierpaolo Pantone, Rob Malouf, and anyone else who may have contributed to the release of the API by reviewing the code and reporting bugs; and anonymous reviewers for their suggestions.

References


