# PyBEL Documentation

***Release 0.11.2-dev***

**Charles Tapley Hoyt**

**Mar 27, 2018**

## Getting Started

Biological Expression Language (BEL) is a domain-specific language that enables the expression of complex molecular relationships and their context in a machine-readable form. Its simple grammar and expressive power have led to its successful use in the IMI project, AETIONOMY, to describe complex disease networks with several thousands of relationships.

PyBEL is a pure Python software package that parses BEL scripts, validates their semantics, and facilitates data interchange between common formats and database systems like JSON, CSV, Excel, SQL, CX, and Neo4J. Its companion package, PyBEL Tools, contains a library of functions for analysis of biological networks. For result-oriented guides, see the PyBEL Notebooks repository.

Installation is as easy as getting the code from PyPI with `python3 -m pip install pybel`

# CHAPTER 1

# Citation

If you use PyBEL in your work, please cite[1]:

[1] Hoyt et al., 2017. PyBEL: a computational framework for Biological Expression Language. Bioinformatics, btx660, https://doi.org/10.1093/bioinformatics/btx660

# CHAPTER 2

# Links

- Specified by BEL 1.0 and BEL 2.0
- Documented on Read the Docs
- Versioned on GitHub
- Tested on Travis CI
- Distributed by PyPI
- Chat on Gitter

## 2.1 Overview

### 2.1.1 Background on Systems Biology Modelling

#### Biological Expression Language (BEL)

Biological Expression Language (BEL) is a domain specific language that enables the expression of complex molecular relationships and their context in a machine-readable form. Its simple grammar and expressive power have led to its successful use in the IMI project, AETIONOMY, to describe complex disease networks with several thousands of relationships. For a detailed explanation, see the BEL 1.0 and 2.0 specifications.

#### OpenBEL Links

- OpenBEL on Google Groups
- OpenBEL Wiki
- OpenBEL on GitHub
- Chat on Gitter

### 2.1.2 Design Considerations

#### Missing Namespaces and Improper Names

The use of openly shared controlled vocabularies (namespaces) within BEL facilitates the exchange and consistency of information. Finding the correct `namespace:name` pair is often a difficult part of the curation process.

#### Outdated Namespaces

OpenBEL provides a variety of namespaces covering each of the BEL function types. These namespaces are generated by code found at https://github.com/OpenBEL/resource-generator and distributed at http://resources.openbel.org/belframework/.

This code has not been maintained to reflect the changes in the underlying resources, so this repository has been forked and updated at https://github.com/pybel/resource-generator to reflect the most recent versions of the underlying namespaces. The files are now distributed using the Fraunhofer SCAI Artifactory server.

#### Generating New Namespaces

In some cases, it is appropriate to design a new namespace, using the custom namespace specification provided by the OpenBEL Framework. Packages for generating namespace, annotation, and knowledge resources have been grouped in the Bio2BEL organization on GitHub.

#### Synonym Issues

Due to the huge number of terms across many namespaces, it's difficult for curators to know the domain-specific synonyms that obscure the controlled/preferred term. However, the issue of synonym resolution and semantic searching has already been generally solved by the use of ontologies. Besides just a controlled vocabulary, they also a hierarchical model of knowledge, synonyms with cross-references to databases and other ontologies, and other information semantic reasoning. Ontologies in the biomedical domain can be found at OBO and EMBL-EBI OLS.

Additionally, as a tool for curators, the EMBL Ontology Lookup Service (OLS) allows for semantic searching. Simple queries for the terms 'mitochondrial dysfunction' and 'amyloid beta-peptides' immediately returned results from relevant ontologies, and ended a long debate over how to represent these objects within BEL. EMBL-EBI also provides a programmatic API to the OLS service, for searching terms (http://www.ebi.ac.uk/ols/api/search?q=folic%20acid) and suggesting resolutions (http://www.ebi.ac.uk/ols/api/suggest?q=folic+acid)

### 2.1.3 Implementation

PyBEL is implemented using the PyParsing module. It provides flexibility and incredible speed in parsing compared to regular expression implementation. It also allows for the addition of parsing action hooks, which allow the graph to be checked semantically at compile-time.

It uses SQLite to provide a consistent and lightweight caching system for external data, such as namespaces, annotations, ontologies, and SQLAlchemy to provide a cross-platform interface. The same data management system is used to store graphs for high-performance querying.

### 2.1.4 Extensions to BEL

The PyBEL compiler is fully compliant with both BEL v1.0 and v2.0 and automatically upgrades legacy statements. Additionally, PyBEL includes several additions to the BEL specification to enable expression of important concepts in molecular biology that were previously missing and to facilitate integrating new data types. A short example is the inclusion of protein oxidation in the default BEL namespace for protein modifications. Other, more elaborate additions are outlined below.

### Syntax for Epigenetics

PyBEL introduces the gene modification function, gmod(), as a syntax for encoding epigenetic modifications. Its usage mirrors the pmod() function for proteins and includes arguments for methylation.

For example, the methylation of NDUFB6 was found to be negatively correlated with its expression in a study of insulin resistance and Type II diabetes. This can now be expressed in BEL such as in the following statement:

```
g(HGNC:NDUFB6, gmod(Me)) negativeCorrelation r(HGNC:NDUFB6)
```

References:

- https://www.ncbi.nlm.nih.gov/pubmed/17948130
- https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4655260/

### Definition of Namespaces as Regular Expressions

BEL imposes the constraint that each identifier must be qualified with an enumerated namespace to enable semantic interoperability and data integration. However, enumerating a namespace with potentially billions of names, such as dbSNP, poses a computational issue. PyBEL introduces syntax for defining namespaces with a consistent pattern using a regular expression to overcome this issue. For these namespaces, semantic validation can be perform in post-processing against the underlying database. The dbSNP namespace can be defined with a syntax familiar to BEL annotation definitions with regular expressions as follows:

```
DEFINE NAMESPACE dbSNP AS PATTERN "rs[0-9]+"
```

### Definition of Resources using OWL

One constraint imposed by the BEL language is that definitions of namespaces and annotations must follow a specific format. However, the creation and maintenance of terminologies in the biological domain has tended towards the usage of the Web Ontology Format (OWL). Services such as the Ontology Lookup Service allow for standardized querying and search of these resources, and provide an important semantic integration layer that previous software tools for BEL did not include. PyBEL allows for these resources to be named directly in definitions with the following syntax:

```
DEFINE ANNOTATION CELL AS OWL "http://purl.obolibrary.org/obo/cl/releases/
2016-11-23/cl.owl" DEFINE NAMESPACE DO AS OWL "http://purl.obolibrary.org/obo/
doid/releases/2017-05-05/doid.owl"
```

This allows PyBEL to import the semantic information from the ontology as well, and provide much more rich algorithms that take into account the hierarchy and synonyms provided.

PyBEL uses the onto2nx package to parse OWL documents in many different formats, including OWL/XML, RDF/XML, and RDF.

### Explicit Node Labels

While the BEL 2.0 specification made it possible to represent new terms, such as the APOE gene with two variants resulting in the E2 allele, it came at the price of encoding terms in a technical and less readable way. An explicit statement for labeling nodes has been added, such that the resulting data structure will have a label for the node:

```
g(HGNC:APOE, var(c.388T>C), var(c.526C>T)) labeled "APOE E2"
```

When InChI is used, these strings are very hard to visualize. Using a label is helpful for later visualization:

```
a(INCHI:"InChI=1S/C20H28N2O5/c1-3-27-20(26)16(12-11-15-8-5-4-6-9-15)21-14(2)18(23)22-13-7-
h4-6,8-9,14,16-17,21H,3,7,10-13H2,1-2H3,(H,24,25)/t14-,16-,17-/m0/s1") labeled
"Enalapril"
```

Below is the same molecule again, but represented with an InChIKey:

```
a(INCHIKEY:"GBXSMTUPTTWBMN-XIRDDKMYSA-N") labeled "Enalapril"
```

It's also easy to use the universe of RESTFul API services from UniChem, ChEMBL, or WikiData to download and annotate these automatically. For futher information on Enalapril can be found WikiData, UniChem, and ChEMBL.

### 2.1.5 Things to Consider

#### Do All Statements Need Supporting Text?

Yes! All statements must be minimally qualified with a citation and evidence (now called SupportingText in BEL 2.0) to maintain provenance. Statements without evidence can't be traced to their source or evaluated independently from the curator, so they are excluded.

#### Multiple Annotations

When an annotation has a list, it means that the following BEL relations are true for each of the listed values. The lines below show a BEL relation that corresponds to two edges, each with the same citation but different values for `ExampleAnnotation`. This should be considered carefully for analyses dealing with the number of edges between two entities.

```
SET Citation = {"PubMed","Example Article","12345"}
SET ExampleAnnotation = {"Example Value 1", "Example Value 2"}
p(HGNC:YFG1) -> p(HGNC:YFG2)
```

Furthermore, if there are multiple annotations with lists, the following BEL relations are true for all of the different combinations of them. The following statements will produce four edges, as the cartesian product of the values used for both `ExampleAnnotation1` and `ExampleAnnotation2`. This might not be the knowledge that the annotator wants to express, and is prone to mistakes, so use of annotation lists are not recommended.

```
SET Citation = {"PubMed","Example Article","12345"}
SET ExampleAnnotation1 = {"Example Value 11", "Example Value 12"}
SET ExampleAnnotation2 = {"Example Value 21", "Example Value 22"}
p(HGNC:YFG1) -> p(HGNC:YFG2)
```

#### Namespace and Annotation Name Choices

`*.belns` and `*.belanno` configuration files include an entry called "Keyword" in their respective [Namespace] and [AnnotationDefinition] sections. To maintain understandability between BEL documents, PyBEL warns when the names given in `*.bel` documents do not match their respective resources. For now, capitalization is not considered, but in the future, PyBEL will also warn when capitalization is not properly stylized, like forgetting the lowercase 'h' in "ChEMBL".

### Why Not Nested Statements?

BEL has different relationships for modeling direct and indirect causal relations.

#### Direct

- `A => B` means that *A* directly increases *B* through a physical process.
- `A =| B` means that *A* directly decreases *B* through a physical process.

#### Indirect

The relationship between two entities can be coded in BEL, even if the process is not well understood.

- `A -> B` means that *A* indirectly increases *B*. There are hidden elements in *X* that mediate this interaction through a pathway direct interactions `A (=> or =|) X_1 (=> or =|) ... X_n (=> or =|) B`, or through a set of multiple pathways that constitute a network.
- `A -| B` means that *A* indirectly decreases *B*. Like for `A -> B`, this process involves hidden components with varying activities.

#### Increasing Nested Relationships

BEL also allows object of a relationship to be another statement.

- `A => (B => C)` means that *A* increases the process by which *B* increases *C*. The example in the BEL Spec `p(HGNC:GATA1) => (act(p(HGNC:ZBTB16)) => r(HGNC:MPL))` represents GATA1 directly increasing the process by which ZBTB16 directly increases MPL. Before, directly increasing was used to specify physical contact, so it's reasonable to conclude that `p(HGNC:GATA1) => act(p(HGNC:ZBTB16))`. The specification cites examples when *B* is an activity that only is affected in the context of *A* and *C*. This complicated enough that it is both impractical to standardize during curation, and impractical to represent in a network.
- `A -> (B => C)` can be interpreted by assuming that *A* indirectly increases *B*, and because of monotonicity, conclude that `A -> C` as well.
- `A => (B -> C)` is more difficult to interpret, because it does not describe which part of process `B -> C` is affected by *A* or how. Is it that `A => B`, and `B => C`, so we conclude `A -> C`, or does it mean something else? Perhaps *A* impacts a different portion of the hidden process in `B -> C`. These statements are ambiguous enough that they should be written as just `A => B`, and `B -> C`. If there is no literature evidence for the statement `A -> C`, then it is not the job of the curator to make this inference. Identifying statements of this might be the goal of a bioinformatics analysis of the BEL network after compilation.
- `A -> (B -> C)` introduces even more ambiguity, and it should not be used.
- `A => (B =| C)` states *A* increases the process by which *B* decreases *C*. One interpretation of this statement might be that `A => B` and `B =| C`. An analysis could infer `A -| C`. Statements in the form of `A -> (B =| C)` can also be resolved this way, but with added ambiguity.

#### Decreasing Nested Relationships

While we could agree on usage for the previous examples, the decrease of a nested statement introduces an unreasonable amount of ambiguity.

- `A =| (B => C)` could mean *A* decreases *B*, and *B* also increases *C*. Does this mean A decreases C, or does it mean that C is still increased, but just not as much? Which of these statements takes precedence? Or do their effects cancel? The same can be said about `A -| (B => C)`, and with added ambiguity for indirect increases `A -| (B -> C)`
- `A =| (B =| C)` could mean that *A* decreases *B* and *B* decreases *C*. We could conclude that *A* increases *C*, or could we again run into the problem of not knowing the precedence? The same is true for the indirect versions.

#### Recommendations for Use in PyBEL

After considering the ambiguity of nested statements to be a great risk to clarity, and PyBEL disables the usage of nested statements by default. See the Input and Output section for different parser settings. At Fraunhofer SCAI, curators resolved these statements to single statements to improve the precision and readability of our BEL documents.

While most statements in the form `A rel1 (B rel2 C)` can be reasonably expanded to `A rel1 B` and `B rel2 C`, the few that cannot are the difficult-to-interpret cases that we need to be careful about in our curation and later analyses.

#### Why Not RDF?

Current bel2rdf serialization tools build URLs with the OpenBEL Framework domain as a namespace, rather than respect the original namespaces of original entities. This does not follow the best practices of the semantic web, where URL's representing an object point to a real page with additional information. For example, UniProt Knowledge Base does an exemplary job of this. Ultimately, using non-standard URL's makes harmonizing and data integration difficult.

Additionally, the RDF format does not easily allow for the annotation of edges. A simple statement in BEL that one protein up-regulates another can be easily represented in a triple in RDF, but when the annotations and citation from the BEL document need to be included, this forces RDF serialization to use approaches like representing the statement itself as a node. RDF was not intended to represent this type of information, but more properly for locating resources (hence its name). Furthermore, many blank nodes are introduced throughout the process. This makes RDF incredibly difficult to understand or work with. Later, writing queries in SPARQL becomes very difficult because the data format is complicated and the language is limited. For example, it would be incredibly complicated to write a query in SPARQL to get the objects of statements from publications by a certain author.

## 2.2 Installation

PyBEL is tested on both Python3 and legacy Python2 installations on Mac OS and Linux using Travis CI as well as on Windows using AppVeyor.

### 2.2.1 Installation

#### Easiest

Download the latest stable code from PyPI with:

```
$ python3 -m pip install pybel
```

#### Get the Latest

Download the most recent code from GitHub with:

```
$ python3 -m pip install git+https://github.com/pybel/pybel.git@develop
```

#### For Developers

Clone the repository from GitHub and install in editable mode with:

```
$ git clone https://github.com/pybel/pybel.git@develop
$ cd pybel
$ python3 -m pip install -e .
```

### 2.2.2 Extras

The `setup.py` makes use of the `extras_require` argument of `setuptools.setup()` in order to make some heavy packages that support special features of PyBEL optional to install, in order to make the installation more lean by default. A single extra can be installed from PyPI like `python3 -m pip install -e pybel[ndex]` or multiple can be installed using a list like `python3 -m pip install -e pybel[ndex,owl]`. Likewise, for developer installation, extras can be installed in editable mode with `python3 -m pip install -e .[ndex]` or multiple can be installed using a list like `python3 -m pip install -e .[ndex,owl]`. The avaliable extras are:

#### ndex

ndex2 supports download and upload to the Network Data Exchange.

**See also:**

- *`pybel.to_ndex()`*
- *`pybel.from_ndex()`*

#### deployment

This extra installs support for Artifactory and enables many of the functions in `pybel.resources`

#### owl

This extra installs support for using OWL ontologies as namespaces via the `onto2nx` package. In the future, this extra will include other wrappers around packages like `obonet` for OBO support.

#### neo4j

This extension installs the `py2neo` package to support upload and download to Neo4j databases.

**See also:**

- *`pybel.to_neo4j()`*

#### indra

This extra installs support for `indra`, the integrated network dynamical reasoner and assembler. Because it also represents biology in BEL-like statements, many statements from PyBEL can be converted to INDRA, and visa-versa. This package also enables the import of BioPAX, SBML, and SBGN into BEL.

**See also:**

- `pybel.from_biopax()`
- `pybel.from_indra_statements()`
- `pybel.from_indra_pickle()`
- `pybel.to_indra()`

### 2.2.3 Caveats

- PyBEL extends the `networkx` for its core data structure. Many of the graphical aspects of `networkx` depend on `matplotlib`, which is an optional dependency.
- If `HTMLlib5` is installed, the test that's supposed to fail on a web page being missing actually tries to parse it as RDFa, and doesn't fail. Disregard this.

### 2.2.4 Upgrading

During the current development cycle, programmatic access to the definition and graph caches might become unstable. If you have any problems working with the database, try removing it with one of the following commands:

1. Running `pybel manage drop` (unix)
2. Running `python3 -m pybel manage drop` (windows)
3. Removing the folder `~/.pybel`

PyBEL will build a new database and populate it on the next run.

## 2.3 Data Model

Molecular biology is a directed graph; not a table. BEL expresses how biological entities interact within many different contexts, with descriptive annotations. PyBEL represents data as a directional multigraph using an extension of `networkx.MultiDiGraph`. Each node and edge has an associated data dictionary for storing relevant/contextual information.

This allows for much easier programmatic access to answer more complicated questions, which can be written with python code. Because the data structure is the same in Neo4J, the data can be directly exported with *`pybel.to_neo4j()`*. Neo4J supports the Cypher querying language so that the same queries can be written in an elegant and simple way.

### 2.3.1 Constants

These documents refer to many aspects of the data model using constants, which can be found in the top-level module *`pybel.constants`*. In these examples, all constants are imported with the following code:

```
>>> from pybel.constants import *
```

Terms describing abundances, annotations, and other internal data are designated in `pybel.constants` with full-caps, such as `pybel.constants.FUNCTION` and `pybel.constants.PROTEIN`.

For normal usage, we suggest referring to values in dictionaries by these constants, in case the hard-coded strings behind these constants change.

### Function Nomenclature

The following table shows PyBEL's internal mapping from BEL functions to its own constants. This can be accessed programatically via `pybel.parser.language.abundance_labels`

| BEL Function | PyBEL Constant |
|---|---|
| `a()`, `abundance()` | `pybel.constants.ABUNDANCE` |
| `g()`, `geneAbundance()` | `pybel.constants.GENE` |
| `r()`, `rnaAbunance()` | `pybel.constants.RNA` |
| `m()`, `microRNAAbundance()` | `pybel.constants.MIRNA` |
| `p()`, `proteinAbundance()` | `pybel.constants.PROTEIN` |
| `bp()`, `biologicalProcess()` | `pybel.constants.BIOPROCESS` |
| `path()`, `pathology()` | `pybel.constants.PATHOLOGY` |
| `complex()`, `complexAbundance()` | `pybel.constants.COMPLEX` |
| `composite()`, `compositeAbundance()` | `pybel.constants.COMPOSITE` |
| `rxn()`, `reaction()` | `pybel.constants.REACTION` |

## 2.3.2 Graph

PyBEL's main data structure is a subclass of `networkx.MultiDiGraph`.

The graph contains metadata for the PyBEL version, the BEL script metadata, the namespace definitions, the annotation definitions, and the warnings produced in analysis. Like any `networkx` graph, all attributes of a given object can be accessed through the `graph` property, like in: `my_graph.graph['my key']`. Convenient property definitions are given for these attributes.

**class** pybel.**BELGraph**(*name=None*, *version=None*, *description=None*, *authors=None*, *contact=None*, *license=None*, *copyright=None*, *disclaimer=None*, *data=None*, ***kwargs*)

This class represents biological knowledge assembled in BEL as a network by extending the `networkx.MultiDiGraph`.

The default constructor parses a BEL graph using the built-in `networkx` methods. For IO, see the `pybel.io` module

**Parameters**

- **name** (*str*) – The graph's name
- **version** (*str*) – The graph's version. Recommended to use semantic versioning or `YYYYMMDD` format.
- **description** (*str*) – A description of the graph
- **authors** (*str*) – The authors of this graph
- **contact** (*str*) – The contact email for this graph
- **license** (*str*) – The license for this graph

- **copyright** (*str*) – The copyright for this graph
- **disclaimer** (*str*) – The disclaimer for this graph
- **data** – initial graph data to pass to `networkx.MultiDiGraph`
- **kwargs** – keyword arguments to pass to `networkx.MultiDiGraph`

**__add__**(*other*)
Creates a deep copy of this graph and full joins another graph with it using *pybel.struct.left_full_join()*.

**Parameters other** (BELGraph) – Another BEL graph

**Return type** *BELGraph*

Example usage:

```
>>> import pybel
>>> g = pybel.from_path('...')
>>> h = pybel.from_path('...')
>>> k = g + h
```

**__iadd__**(*other*)
Full joins another graph into this one using *pybel.struct.left_full_join()*.

**Parameters other** (BELGraph) – Another BEL graph

**Return type** *BELGraph*

Example usage:

```
>>> import pybel
>>> g = pybel.from_path('...')
>>> h = pybel.from_path('...')
>>> g += h
```

**__and__**(*other*)
Creates a deep copy of this graph and outer joins another graph with it using *pybel.struct.left_outer_join()*.

**Parameters other** (BELGraph) – Another BEL graph

**Return type** *BELGraph*

Example usage:

```
>>> import pybel
>>> g = pybel.from_path('...')
>>> h = pybel.from_path('...')
>>> k = g & h
```

**__iand__**(*other*)
Outer joins another graph into this one using *pybel.struct.left_outer_join()*.

**Parameters other** (BELGraph) – Another BEL graph

**Return type** *BELGraph*

Example usage:

```
>>> import pybel
>>> g = pybel.from_path('...')
>>> h = pybel.from_path('...')
>>> g &= h
```

**document**
A dictionary holding the metadata from the "Document" section of the BEL script. All keys are normalized according to *pybel.constants.DOCUMENT_KEYS*

**Return type** dict[str,str]

**name**
The graph's name, from the `SET DOCUMENT Name = "..."` entry in the source BEL script

**Return type** str

**version**
The graph's version, from the `SET DOCUMENT Version = "..."` entry in the source BEL script

**Return type** str

**description**
The graph's description, from the `SET DOCUMENT Description = "..."` entry in the source BEL Script

**Return type** str

**authors**
The graph's description, from the `SET DOCUMENT Authors = "..."` entry in the source BEL Script

**Return type** str

**contact**
The graph's description, from the `SET DOCUMENT ContactInfo = "..."` entry in the source BEL Script

**Return type** str

**license**
The graph's license, from the *SET DOCUMENT Licenses* = "… "' entry in the source BEL Script

**Return type** Optional[str]

**copyright**
The graph's copyright, from the *SET DOCUMENT Copyright* = "… "' entry in the source BEL Script

**Return type** Optional[str]

**disclaimer**
The graph's disclaimer, from the *SET DOCUMENT Disclaimer* = "… "' entry in the source BEL Script

**Return type** Optional[str]

**namespace_url**
A dictionary mapping the keywords used to create this graph to the URLs of the BELNS files from the `DEFINE NAMESPACE [key] AS URL "[value]"` entries in the definitions section.

**Return type** dict[str,str]

**namespace_owl**
A dictionary mapping the keywords used to create this graph to the URLs of the OWL files from the `DEFINE NAMESPACE [key] AS OWL "[value]"` entries in the definitions section

**Return type** dict[str,str]

**defined_namespace_keywords**
Returns the set of all keywords defined as namespaces in this graph

**Return type** set[str]

**uncached_namespaces**
Returns a list of namespaces's URLs that are present in the graph, but cannot be cached due to their corresponding resources' cachable flags being set to "no."

**Return type** set[str]

**namespace_pattern**
A dictionary mapping the namespace keywords used to create this graph to their regex patterns from the `DEFINE NAMESPACE [key] AS PATTERN "[value]"` entries in the definitions section

**Return type** dict[str,str]

**annotation_url**
A dictionary mapping the annotation keywords used to create this graph to the URLs of the BELANNO files from the `DEFINE ANNOTATION [key] AS URL "[value]"` entries in the definitions section

**Return type** dict[str,str]

**annotation_owl**
A dictionary mapping the annotation keywords used to creat ethis graph to the URLs of the OWL files from the `DEFINE ANNOTATION [key] AS OWL "[value]"` entries in the definitions section

**Return type** dict[str,str]

**annotation_pattern**
A dictionary mapping the annotation keywords used to create this graph to their regex patterns from the `DEFINE ANNOTATION [key] AS PATTERN "[value]"` entries in the definitions section

**Return type** dict[str,str]

**annotation_list**
A dictionary mapping the keywords of locally defined annotations to a set of their values from the `DEFINE ANNOTATION [key] AS LIST {"[value]", ...}` entries in the definitions section

**Return type** dict[str,set[str]]

**defined_annotation_keywords**
Returns the set of all keywords defined as annotations in this graph

**Return type** set[str]

**pybel_version**
Stores the version of PyBEL with which this graph was produced as a string

**Return type** str

**warnings**
Warnings are stored in a list of 4-tuples that is a property of the graph object. This tuple respectively contains the line number, the line text, the exception instance, and the context dictionary from the parser at the time of error.

**Return type** list[tuple[int,str,Exception,dict[str,str]]]

**skip_storing_namespace**(*namespace*)
Checks if the namespace should be skipped

**Parameters** **namespace** (*Optional[str]*) –

**Return type** bool

**add_unqualified_edge** (*u*, *v*, *relation*)

Adds unique edge that has no annotations

**Parameters**

- **u** (*tuple or dict*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the source node
- **v** (*tuple or dict*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the target node
- **relation** (*str*) – A relationship label from `pybel.constants`

**Returns** The hash of this edge

**Return type** str

**add_transcription** (*u*, *v*)

Adds a transcription relation from a gene to an RNA or miRNA node

**Parameters**

- **u** (*tuple or dict*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the source node
- **v** (*tuple or dict*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the target node

**add_translation** (*u*, *v*)

Adds a translation relation from a RNA to a protein

**Parameters**

- **u** (*tuple or dict*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the source node
- **v** (*tuple or dict*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the target node

**add_equivalence** (*u*, *v*)

Adds two equivalence relations for the nodes

**Parameters**

- **u** (*tuple or dict*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the source node
- **v** (*tuple or dict*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the target node

**add_orthology** (*u*, *v*)

Adds two orthology relations for the nodes

**Parameters**

- **u** (*tuple or dict*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the source node
- **v** (*tuple or dict*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the target node

**add_is_a** (*u*, *v*)

Adds an isA relationship such that u isA v.

**Parameters**

- **u** (*tuple or dict*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the source node
- **v** (*tuple or dict*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the target node

**iter_node_data_pairs**()
Iterates over pairs of nodes and their data dictionaries

**Return type** iter[tuple[tuple,dict]]

**iter_data**()
Iterates over the node data dictionaries

**Return type** iter[dict]

**add_node_from_data**(*attr_dict*)
Converts a PyBEL node data dictionary to a canonical PyBEL node tuple and ensures it is in the graph.

**Parameters** **attr_dict** (*dict*) – A PyBEL node data dictionary

**Returns** A PyBEL node tuple

**Return type** tuple

**has_node_with_data**(*attr_dict*)
Checks if this graph has a node with the given data dictionary

**Parameters** **attr_dict** (*dict*) – A PyBEL node data dictionary

**Return type** bool

**add_simple_node**(*func*, *namespace*, *name*)
Adds a simple node, with only a namespace and name

**Parameters**

- **func** (*str*) – The node's function (*pybel.constants.GENE*, *pybel.constants.PROTEIN*, etc)
- **namespace** (*str*) – The node's namespace
- **name** (*str*) – The node's name

**Returns** The PyBEL node tuple representing this node

**Return type** tuple

**add_qualified_edge**(*u*, *v*, *relation*, *evidence*, *citation*, *annotations=None*, *subject_modifier=None*, *object_modifier=None*, ***attr*)
Adds an edge, qualified with a relation, evidence, citation, and optional annotations, subject modifications, and object modifications

**Parameters**

- **or dict u** (*tuple*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the source node
- **or dict v** (*tuple*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the target node
- **relation** (*str*) – The type of relation this edge represents
- **evidence** (*str*) – The evidence string from an article

- **or str citation** (*dict[str,str]*) – The citation data dictionary for this evidence. If a string is given, assumes it's a PubMed identifier and auto-fills the citation type.
- **annotations** (*Optional[dict[str,str] or dict[str,set] or dict[str,dict[str,bool]]]*) – The annotations data dictionary
- **subject_modifier** (*Optional[dict]*) – The modifiers (like activity) on the subject node. See data model documentation.
- **object_modifier** (*Optional[dict]*) – The modifiers (like activity) on the object node. See data model documentation.

**Returns** The hash of the edge

**Return type** str

**add_inhibits**(*u*, *v*, *evidence*, *citation*, *annotations=None*, *object_modifier=None*)
A more specific version of add_qualified edge that automatically populates the relation and object modifier

**Parameters**

- **or dict u** (*tuple*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the source node
- **or dict v** (*tuple*) – Either a PyBEL node tuple or PyBEL node data dictionary representing the target node
- **evidence** (*str*) – The evidence string from an article
- **or str citation** (*dict[str,str]*) – The citation data dictionary for this evidence. If a string is given, assumes it's a PubMed identifier and autofills the citation type.
- **annotations** (*Optional[dict[str,str] or dict[str,set] or dict[str,dict[str,bool]]]*) – The annotations data dictionary
- **object_modifier** (*Optional[dict]*) – A non-default activity.

**Returns** The hash of the edge

**Return type** str

**has_edge_citation**(*u*, *v*, *key*)
Does the given edge have a citation?

**Return type** bool

**get_edge_citation**(*u*, *v*, *key*)
Gets the citation for a given edge

**Return type** Optional[dict]

**has_edge_evidence**(*u*, *v*, *key*)
Does the given edge have evidence?

**Return type** boolean

**get_edge_evidence**(*u*, *v*, *key*)
Gets the evidence for a given edge

**Return type** Optional[str]

**get_edge_annotations**(*u*, *v*, *key*)
Gets the annotations for a given edge

**Return type** Optional[dict]

**get_node_name**(*node*)
Gets the node's name, or return None if no name

**Return type** Optional[str]

**set_node_name**(*node*, *name*)
Sets the name for a given node

**Parameters** **node** (*tuple*) – A PyBEL node tuple

**get_node_identifier**(*node*)
Gets the identifier for a given node from the database (not the same as the node hash)

**Return type** Optional[str]

**get_node_description**(*node*)
Gets the description for a given node

**Return type** Optional[str]

**has_node_description**(*node*)
Returns if a node description is already present

**Parameters** **node** (*tuple*) – A PyBEL node tuple

**Return type** bool

**set_node_description**(*node*, *description*)
Sets the description for a given node

**Parameters** **node** (*tuple*) – A PyBEL node tuple

**node_to_bel**(*n*)
Serializes a node as BEL

**Parameters** **n** (*tuple*) – A PyBEL node tuple

**Return type** str

**edge_to_bel**(*u*, *v*, *data*, *sep=None*)
Serializes a pair of nodes and related edge data as a BEL relation

**Parameters**

- **u** (*tuple*) – A PyBEL node tuple for the soure node
- **v** (*tuple*) – A PyBEL node tuple for the target node
- **data** (*dict*) – A PyBEL edge data dictionary
- **sep** (*str*) – The separator between the source, relation, and target. Defaults to ' '

**Return type** str

**iter_equivalent_nodes**(*node*)
Iterates over node tuples that are equivalent to the given node, including the original

**Parameters** **node** (*tuple*) – A PyBEL node tuple

**Return type** iter[tuple]

**get_equivalent_nodes**(*node*)
Gets a set of equivalent nodes to this node. Does not include the given node.

**Parameters** **node** (*tuple*) – A PyBEL node tuple

**Return type** set[tuple]

**node_has_namespace** (*node*, *namespace*)
Does the node have the given namespace? This also should look in the equivalent nodes.

**Parameters**

- **node** (*tuple*) – A PyBEL node tuple
- **namespace** (*str*) – A namespace

**Return type** bool

pybel.struct.**left_full_join** (*g*, *h*, *use_hash=True*)
Adds all nodes and edges from h to g, in-place for g

**Parameters**

- **g** (BELGraph) – A BEL network
- **h** (BELGraph) – A BEL network
- **use_hash** (*bool*) – If true, uses a hash join algorithm. Else, uses an exhaustive search, which takes much longer.

Example usage:

```
>>> import pybel
>>> g = pybel.from_path('...')
>>> h = pybel.from_path('...')
>>> merged = left_full_join(g, h)
```

pybel.struct.**left_outer_join** (*g*, *h*, *use_hash=True*)
Only adds components from the h that are touching g.

Algorithm:

1. Identify all weakly connected components in h
2. Add those that have an intersection with the g

**Parameters**

- **g** (BELGraph) – A BEL network
- **h** (BELGraph) – A BEL network
- **use_hash** (*bool*) – If true, uses a hash join algorithm. Else, uses an exhaustive search, which takes much longer.

Example usage:

```
>>> import pybel
>>> g = pybel.from_path('...')
>>> h = pybel.from_path('...')
>>> merged = left_outer_join(g, h)
```

pybel.struct.**union** (*networks*, *use_hash=True*)
Takes the union over a collection of networks into a new network. Assumes iterator is longer than 2, but not infinite.

**Parameters**

- **networks** (*iter[BELGraph]*) – An iterator over BEL networks. Can't be infinite.

- **use_hash** (*bool*) – If true, uses a hash join algorithm. Else, uses an exhaustive search, which takes much longer.

**Returns** A merged network

**Return type** *BELGraph*

Example usage:

```
>>> import pybel
>>> g = pybel.from_path('...')
>>> h = pybel.from_path('...')
>>> k = pybel.from_path('...')
>>> merged = union([g, h, k])
```

## 2.3.3 Nodes

Nodes are used to represent physical entities' abundances. The relevant data about a node is stored in its associated data dictionary in `networkx` that can be accessed with `my_bel_graph.node[node]`. After parsing, `p(HGNC:GSK3B)` becomes:

```
{
    FUNCTION: PROTEIN,
    NAMESPACE: 'HGNC',
    NAME: 'GSK3B'
}
```

This section describes the structure of the data dictionaries created for each type of node available in BEL. Programatically, these dictionaries can be converted to tuples, which are used as the keys for the network with the `pybel.parser.canonicalize.node_to_tuple()` function.

### Variants

The addition of a variant tag results in an entry called 'variants' in the data dictionary associated with a given node. This entry is a list with dictionaries describing each of the variants. All variants have the entry 'kind' to identify whether it is a post-translational modification (PTM), gene modification, fragment, or HGVS variant.

**Warning:** The canonical ordering for the elements of the `VARIANTS` list correspond to the sorted order of their corresponding node tuples using `pybel.parser.canonicalize.sort_dict_list()`. Rather than directly modifying the BELGraph's structure, use *`pybel.BELGraph.add_node_from_data()`*, which takes care of automatically canonicalizing this dictionary.

### HGVS Variants

For example, the node `p(HGNC:GSK3B, var(p.Gly123Arg))` is represented with the following:

```
{
     FUNCTION: PROTEIN,
     NAMESPACE: 'HGNC',
     NAME: 'GSK3B',
     VARIANTS:  [
         {
            KIND: HGVS,
            IDENTIFIER: 'p.Gly123Arg'
         }
    ]
}
```

**See also:**

- BEL 2.0 specification on variants
- HVGS conventions
- PyBEL module `pybel.parser.modifiers.VariantParser`

### Gene Substitution

Gene substitutions are legacy statements defined in BEL 1.0. BEL 2.0 recommends using HGVS strings. Luckily, the information contained in a BEL 1.0 encoding, such as `g(HGNC:APP,sub(G,275341,C))` can be automatically translated to the appropriate HGVS `g(HGNC:APP, var(c.275341G>C))`, assuming that all substitutions are using the reference coding gene sequence for numbering and not the genomic reference. The previous statements both produce the underlying data:

```
{
    FUNCTION: GENE,
    NAMESPACE: 'HGNC',
    NAME: 'APP',
    VARIANTS: [
        {
            KIND: HGVS,
            IDENTIFIER: 'c.275341G>C'
        }
    ]
}
```

**See also:**

- BEL 2.0 specification on gene substitutions
- PyBEL module `pybel.parser.modifiers.GeneSubstitutionParser`

### Gene Modification

PyBEL introduces the gene modification tag, gmod(), to allow for the encoding of epigenetic modifications. Its syntax follows the same style s the pmod() tags for proteins, and can include the following values:

- M
- Me
- methylation
- A
- Ac
- acetylation

For example, the node `g(HGNC:GSK3B, gmod(M))` is represented with the following:

```
{
    FUNCTION: GENE,
    NAMESPACE: 'HGNC',
    NAME: 'GSK3B',
    VARIANTS: [
        {
            KIND: GMOD,
            IDENTIFIER: {
                NAMESPACE: BEL_DEFAULT_NAMESPACE,
                NAME: 'Me'
            }
        }
    ]
}
```

The addition of this function does not preclude the use of all other standard functions in BEL; however, other compilers probably won't support these standards. If you agree that this is useful, please contribute to discussion in the OpenBEL community.

**See also:**

- PyBEL module `pybel.parser.modifiers.GeneModificationParser`

### Protein Substitution

Protein substitutions are legacy statements defined in BEL 1.0. BEL 2.0 recommends using HGVS strings. Luckily, the information contained in a BEL 1.0 encoding, such as `p(HGNC:APP,sub(R,275,H))` can be automatically translated to the appropriate HGVS `p(HGNC:APP, var(p.Arg275His))`, assuming that all substitutions are using the reference protein sequence for numbering and not the genomic reference. The previous statements both produce the underlying data:

```
{
    FUNCTION: GENE,
    NAMESPACE: 'HGNC',
    NAME: 'APP',
    VARIANTS: [
        {
            KIND: HGVS,
            IDENTIFIER: 'p.Arg275His'
        }
    ]
}
```

**See also:**

- BEL 2.0 specification on protein substitutions
- PyBEL module `pybel.parser.modifiers.ProteinSubstitutionParser`

### Protein Modification

The addition of a post-translational modification (PTM) tag results in an entry called 'variants' in the data dictionary associated with a given node. This entry is a list with dictionaries describing each of the variants. All variants have the entry 'kind' to identify whether it is a PTM, gene modification, fragment, or HGVS variant. The 'kind' value for PTM is 'pmod'.

Each PMOD contains an identifier, which is a dictionary with the namespace and name, and can optionally include the position ('pos') and/or amino acid code ('code').

For example, the node `p(HGNC:GSK3B, pmod(P, S, 9))` is represented with the following:

```
{
    FUNCTION: PROTEIN,
    NAMESPACE: 'HGNC',
    NAME: 'GSK3B',
    VARIANTS: [
        {
            KIND: PMOD,
            IDENTIFIER: {
                NAMESPACE: BEL_DEFAULT_NAMESPACE
                NAME: 'Ph',

            },
            PMOD_CODE: 'Ser',
            PMOD_POSITION: 9
        }
    ]
}
```

As an additional example, in `p(HGNC:MAPK1, pmod(Ph, Thr, 202), pmod(Ph, Tyr, 204))`, MAPK is phosphorylated twice to become active. This results in the following:

```
{
    FUNCTION: PROTEIN,
    NAMESPACE: 'HGNC',
    NAME: 'MAPK1',
    VARIANTS: [
        {
            KIND: PMOD,
            IDENTIFIER: {
                NAMESPACE: BEL_DEFAULT_NAMESPACE
                NAME: 'Ph',

            },
            PMOD_CODE: 'Thr',
            PMOD_POSITION: 202
        },
        {
            KIND: PMOD,
            IDENTIFIER: {
                NAMESPACE: BEL_DEFAULT_NAMESPACE
                NAME: 'Ph',

            },
            PMOD_CODE: 'Tyr',
            PMOD_POSITION: 204
        }
    ]
}
```

**See also:**

- BEL 2.0 specification on protein modifications
- PyBEL module `pybel.parser.modifiers.ProteinModificationParser`

### Truncations

Truncations in the legacy BEL 1.0 specification are automatically translated to BEL 2.0 with HGVS nomenclature. `p(HGNC:AKT1, trunc(40))` becomes `p(HGNC:AKT1, var(p.40*))` and is represented with the following dictionary:

```
{
    FUNCTION: PROTEIN,
    NAMESPACE: 'HGNC',
    NAME: 'AKT1',
    VARIANTS: [
        {
            KIND: HGVS,
            IDENTIFIER: 'p.40*'
        }
    ]
}
```

Unfortunately, the HGVS nomenclature requires the encoding of the terminal amino acid which is exchanged for a stop codon, and this information is not required by BEL 1.0. For this example, the proper encoding of the truncation at position also includes the information that the 40th amino acid in the AKT1 is Cys. Its BEL encoding should be `p(HGNC:AKT1, var(p.Cys40*))`. Temporary support has been added to compile these statements, but it's recommended they are upgraded by reexamining the supporting text, or looking up the amino acid sequence.

**See also:**

- BEL 2.0 specification on truncations
- PyBEL module *`pybel.parser.modifiers.TruncationParser`*

### Fragments

The addition of a fragment results in an entry called *`pybel.constants.VARIANTS`* in the data dictionary associated with a given node. This entry is a list with dictionaries describing each of the variants. All variants have the entry *`pybel.constants.KIND`* to identify whether it is a PTM, gene modification, fragment, or HGVS variant. The *`pybel.constants.KIND`* value for a fragment is *`pybel.constants.FRAGMENT`*.

Each fragment contains an identifier, which is a dictionary with the namespace and name, and can optionally include the position ('pos') and/or amino acid code ('code').

For example, the node `p(HGNC:GSK3B, frag(45_129))` is represented with the following:

```
{
    FUNCTION: PROTEIN,
    NAMESPACE: 'HGNC',
    NAME: 'GSK3B',
    VARIANTS: [
        {
            KIND: FRAGMENT,
            FRAGMENT_START: 45,
            FRAGMENT_STOP: 129
        }
    ]
}
```

Additionally, nodes can have an asterick (*) or question mark (?) representing unbound or unknown fragments, respectively.

A fragment may also be unknown, such as in the node `p(HGNC:GSK3B, frag(?))`. This is represented with the key *`pybel.constants.FRAGMENT_MISSING`* and the value of '?' like:

```
{
    FUNCTION: PROTEIN,
    NAMESPACE: 'HGNC',
    NAME: 'GSK3B',
    VARIANTS: [
        {
            KIND: FRAGMENT,
            FRAGMENT_MISSING: '?',
        }
    ]
}
```

**See also:**

- BEL 2.0 specification on proteolytic fragments (2.2.3)
- PyBEL module `pybel.parser.modifiers.FragmentParser`

### Fusions

### Fusions

Gene, RNA, protein, and miRNA fusions are all represented with the same underlying data structure. Below it is shown with uppercase letters referring to constants from `pybel.constants` and. For example, `g(HGNC:BCR, fus(HGNC:JAK2, 1875, 2626))` is represented as:

```
{
    FUNCTION: GENE,
    FUSION: {
        PARTNER_5P: {NAMESPACE: 'HGNC', NAME: 'BCR'},
        PARTNER_3P: {NAMESPACE: 'HGNC', NAME: 'JAK2'},
        RANGE_5P: {
            FUSION_REFERENCE: 'c',
            FUSION_START: '?',
            FUSION_STOP: 1875

        },
        RANGE_3P: {
            FUSION_REFERENCE: 'c',
            FUSION_START: 2626,
            FUSION_STOP: '?'
        }
    }
}
```

**See also:**

- BEL 2.0 specification on fusions (2.6.1)
- PyBEL module *`pybel.parser.modifiers.FusionParser`*

## 2.3.4 Unqualified Edges

Unqualified edges are automatically inferred by PyBEL and do not contain citations or supporting evidence.

### Variant and Modifications' Parent Relations

All variants, modifications, fragments, and truncations are connected to their parent entity with an edge having the relationship `hasParent`

For `p(HGNC:GSK3B, var(p.Gly123Arg))`, the following edge is inferred:

```
p(HGNC:GSK3B, var(p.Gly123Arg)) hasParent p(HGNC:GSK3B)
```

All variants have this relationship to their reference node. BEL does not specify relationships between variants, such as the case when a given phosphorylation is necessary to make another one. This knowledge could be encoded directly like BEL, since PyBEL does not restrict users from manually asserting unqualified edges.

### List Abundances

Complexes and composites that are defined by lists. As of version 0.9.0, they contain a list of the data dictionaries that describe their members. For example `complex(p(HGNC:FOS), p(HGNC:JUN))` becomes:

```
{
    FUNCTION: COMPLEX,
    MEMBERS: [
        {
            FUNCTION: PROTEIN,
            NAMESPACE: 'HGNC',
            NAME: 'FOS'
        }, {
            FUNCTION: PROTEIN,
            NAMESPACE: 'HGNC',
            NAME: 'JUN'
        }
    ]
}
```

The following edges are also inferred:

```
complex(p(HGNC:FOS), p(HGNC:JUN)) hasMember p(HGNC:FOS)
complex(p(HGNC:FOS), p(HGNC:JUN)) hasMember p(HGNC:JUN)
```

**See also:**

BEL 2.0 specification on complex abundances

Similarly, `composite(a(CHEBI:malonate), p(HGNC:JUN))` becomes:

```
{
    FUNCTION: COMPOSITE,
    MEMBERS: [
        {
            FUNCTION: ABUNDANCE,
            NAMESPACE: 'CHEBI',
            NAME: 'malonate'
        }, {
            FUNCTION: PROTEIN,
            NAMESPACE: 'HGNC',
            NAME: 'JUN'
        }
    ]
}
```

The following edges are inferred:

```
composite(a(CHEBI:malonate), p(HGNC:JUN)) hasComponent a(CHEBI:malonate)
composite(a(CHEBI:malonate), p(HGNC:JUN)) hasComponent p(HGNC:JUN)
```

**Warning:** The canonical ordering for the elements of the `MEMBERS` list correspond to the sorted order of their corresponding node tuples using `pybel.parser.canonicalize.sort_dict_list()`. Rather than directly modifying the BELGraph's structure, use `BELGraph.add_node_from_data()`, which takes care of automatically canonicalizing this dictionary.

**See also:**

BEL 2.0 specification on composite abundances

### Reactions

The usage of a reaction causes many nodes and edges to be created. The following example will illustrate what is added to the network for

```
rxn(reactants(a(CHEBI:"(3S)-3-hydroxy-3-methylglutaryl-CoA"), a(CHEBI:"NADPH"), \
    a(CHEBI:"hydron")), products(a(CHEBI:"mevalonate"), a(CHEBI:"NADP(+)")))
```

As of version 0.9.0, the reactants' and products' data dictionaries are included as sub-lists keyed `REACTANTS` and `PRODUCTS`. It becomes:

```
{
    FUNCTION: REACTION
    REACTANTS: [
        {
            FUNCTION: ABUNDANCE,
            NAMESPACE: 'CHEBI',
            NAME: '(3S)-3-hydroxy-3-methylglutaryl-CoA'
        }, {
            FUNCTION: ABUNDANCE,
            NAMESPACE: 'CHEBI',
            NAME: 'NADPH'
        }, {
            FUNCTION: ABUNDANCE,
            NAMESPACE: 'CHEBI',
            NAME: 'hydron'
        }
    ],
    PRODUCTS: [
        {
            FUNCTION: ABUNDANCE,
            NAMESPACE: 'CHEBI',
            NAME: 'mevalonate'
        }, {
            FUNCTION: ABUNDANCE,
            NAMESPACE: 'CHEBI',
            NAME: 'NADP(+)'
        }
    ]
}
```

**Warning:** The canonical ordering for the elements of the `REACTANTS` and `PRODUCTS` lists correspond to the sorted order of their corresponding node tuples using `pybel.parser.canonicalize.sort_dict_list()`. Rather than directly modifying the BELGraph's structure, use `BELGraph.add_node_from_data()`, which takes care of automatically canonicalizing this dictionary.

The following edges are inferred, where `X` represents the previous reaction, for brevity:

```
X hasReactant a(CHEBI:"(3S)-3-hydroxy-3-methylglutaryl-CoA")
X hasReactant a(CHEBI:"NADPH")
X hasReactant a(CHEBI:"hydron")
X hasProduct a(CHEBI:"mevalonate")
X hasProduct a(CHEBI:"NADP(+)"))
```

**See also:**

BEL 2.0 specification on reactions

### 2.3.5 Edges

#### Design Choices

In the OpenBEL Framework, modifiers such as activities (kinaseActivity, etc.) and transformations (translocations, degradations, etc.) were represented as their own nodes. In PyBEL, these modifiers are represented as a property of the edge. In reality, an edge like `sec(p(HGNC:A)) -> activity(p(HGNC:B), ma(kinaseActivity))` represents a connection between `HGNC:A` and `HGNC:B`. Each of these modifiers explains the context of the relationship between these physical entities. Further, querying a network where these modifiers are part of a relationship is much more straightforward. For example, finding all proteins that are upregulated by the kinase activity of another protein now can be directly queried by filtering all edges for those with a subject modifier whose modification is molecular activity, and whose effect is kinase activity. Having fewer nodes also allows for a much easier display and visual interpretation of a network. The information about the modifier on the subject and activity can be displayed as a color coded source and terminus of the connecting edge.

The compiler in OpenBEL framework created nodes for molecular activities like `kin(p(HGNC:YFG))` and induced an edge like `p(HGNC:YFG) actsIn kin(p(HGNC:YFG))`. For transformations, a statement like `tloc(p(HGNC:YFG), GOCC:intracellular, GOCC:"cell membrane")` also induced `tloc(p(HGNC:YFG), GOCC:intracellular, GOCC:"cell membrane") translocates p(HGNC:YFG)`.

In PyBEL, we recognize that these modifications are actually annotations to the type of relationship between the subject's entity and the object's entity. `p(HGNC:ABC) -> tloc(p(HGNC:YFG), GOCC:intracellular, GOCC:"cell membrane")` is about the relationship between `p(HGNC:ABC)` and `p(HGNC:YFG)`, while the information about the translocation qualifies that the object is undergoing an event, and not just the abundance. This is a confusion with the use of `proteinAbundance` as a keyword, and perhaps is why many people prefer to use just the keyword `p`

#### Example Edge Data Structure

Because this data is associated with an edge, the node data for the subject and object are not included explicitly. However, information about the activities, modifiers, and transformations on the subject and object are included. Below is the "skeleton" for the edge data model in PyBEL:

```
{
    SUBJECT: {
        # ... modifications to the subject node. Only present if non-empty.
    },
    RELATION: POSITIVE_CORRELATION,
    OBJECT: {
        # ... modifications to the object node. Only present if non-empty.
    },
    EVIDENCE: '...',
    CITATION : {
        CITATION_TYPE: CITATION_TYPE_PUBMED,
        CITATION_REFERENCE: '...',
        CITATION_DATE: 'YYYY-MM-DD',
        CITATION_AUTHORS: 'Jon Snow|John Doe',
    },
    ANNOTATIONS: {
        'Disease': {
            'Colorectal Cancer': True,
         }
        # ... additional annotations as tuple[str,dict[str,bool]] pairs
    }
}
```

Each edge must contain the `RELATION`, `EVIDENCE`, and `CITATION` entries. The `CITATION` must minimally contain `CITATION_TYPE` and `CITATION_REFERENCE` since these can be used to look up additional metadata.

**Note:** Since version 0.10.2, annotations now always appear as dictionaries, even if only one value is present.

### Activities

Modifiers are added to this structure as well. Under this schema, `p(HGNC:GSK3B, pmod(P, S, 9)) pos act(p(HGNC:GSK3B), ma(kin))` becomes:

```
{
    RELATION: POSITIVE_CORRELATION,
    OBJECT: {
        MODIFIER: ACTIVITY,
        EFFECT: {
            NAME: 'kin',
            NAMESPACE: BEL_DEFAULT_NAMESPACE
        }
    },
    CITATION: { ... },
    EVIDENCE: '...',
    ANNOTATIONS: { ... }
}
```

Activities without molecular activity annotations do not contain an `pybel.constants.EFFECT` entry: Under this schema, `p(HGNC:GSK3B, pmod(P, S, 9)) pos act(p(HGNC:GSK3B))` becomes:

```
{
    RELATION: POSITIVE_CORRELATION,
    OBJECT: {
        MODIFIER: ACTIVITY
    },
    CITATION: { ... },
    EVIDENCE: '...',
    ANNOTATIONS: { ... }
}
```

### Locations

Location data also is added into the information in the edge for the node (subject or object) for which it was annotated. `p(HGNC:GSK3B, pmod(P, S, 9), loc(GOCC:lysozome)) pos act(p(HGNC:GSK3B), ma(kin))` becomes:

```
{
    SUBJECT: {
        LOCATION: {
            NAMESPACE: 'GOCC',
            NAME: 'lysozome'
        }
    },
    RELATION: POSITIVE_CORRELATION,
    OBJECT: {
        MODIFIER: ACTIVITY,
        EFFECT: {
            NAMESPACE: BEL_DEFAULT_NAMESPACE
            NAME: 'kin',
        }
    },
    EVIDENCE: '...',
    CITATION: { ... }
}
```

The addition of the `location()` element in BEL 2.0 allows for the unambiguous expression of the differences between the process of hypothetical `HGNC:A` moving from one place to another and the existence of hypothetical `HGNC:A` in a specific location having different effects. In BEL 1.0, this action had its own node, but this introduced unnecessary complexity to the network and made querying more difficult. This calls for thoughtful consideration of the following two statements:

- `tloc(p(HGNC:A), fromLoc(GOCC:intracellular), toLoc(GOCC:"cell membrane")) -> p(HGNC:B)`
- `p(HGNC:A, location(GOCC:"cell membrane")) -> p(HGNC:B)`

**See also:**

- BEL 2.0 specification on cellular location (2.2.4)
- PyBEL module `pybel.parser.modifiers.LocationParser`

### Translocations

Translocations have their own unique syntax. `p(HGNC:YFG1) -> sec(p(HGNC:YFG2))` becomes:

```
{
    RELATION: INCREASES,
    OBJECT: {
        MODIFIER: TRANSLOCATION,
        EFFECT: {
            FROM_LOC: {
                NAMESPACE: 'GOCC',
                NAME: 'intracellular'
            },
            TO_LOC: {
                NAMESPACE: 'GOCC',
                NAME: 'extracellular space'
            }
        }
    },
    CITATION: { ... },
    EVIDENCE: '...',
    ANNOTATIONS: { ... }
}
```

**See also:**

BEL 2.0 specification on translocations

### Degradations

Degradations are more simple, because there's no :pybel.constants.EFFECT entry. `p(HGNC:YFG1) -> deg(p(HGNC:YFG2))` becomes:

```
{
    RELATION: INCREASES,
    OBJECT: {
        MODIFIER: DEGRADATION
    },
    CITATION: { ... },
    EVIDENCE: '...',
    ANNOTATIONS: { ... }
}
```

**See also:**

BEL 2.0 specification on degradations

## 2.4 Example Networks

This directory contains example networks, precompiled as BEL graphs that are appropriate to use in examples. The following is an example on EGF's effect on cellular processes

```
SET Citation = {"PubMed","Clin Cancer Res 2003 Jul 9(7) 2416-25","12855613"}
SET Evidence = "This induction was not seen either when LNCaP cells were treated with␣
↪flutamide or conditioned medium were pretreated with antibody to the epidermal␣
↪growth factor (EGF)"
SET Species = 9606

tscript(p(HGNC:AR)) increases p(HGNC:EGF)

UNSET ALL

SET Citation = {"PubMed","Int J Cancer 1998 Jul 3 77(1) 138-45","9639405"}
SET Evidence = "DU-145 cells treated with 5000 U/ml of IFNgamma and IFN alpha, both␣
↪reduced EGF production with IFN gamma reduction more significant."
SET Species = 9606
```

```
p(HGNC:IFNA1) decreases p(HGNC:EGF)
p(HGNC:IFNG) decreases p(HGNC:EGF)

UNSET ALL


SET Citation = {"PubMed","DNA Cell Biol 2000 May 19(5) 253-63","10855792"}
SET Evidence = "Although found predominantly in the cytoplasm and, less abundantly,␣
↪in the nucleus, VCP can be translocated from the nucleus after stimulation with␣
↪epidermal growth factor."
SET Species = 9606

p(HGNC:EGF) increases tloc(p(HGNC:VCP),GOCCID:0005634,GOCCID:0005737)

UNSET ALL

SET Citation = {"PubMed","J Clin Oncol 2003 Feb 1 21(3) 447-52","12560433"}
SET Evidence = "Valosin-containing protein (VCP; also known as p97) has been shown to␣
↪be associated with antiapoptotic function and metastasis via activation of the␣
↪nuclear factor-kappaB signaling pathway."
SET Species = 9606

cat(p(HGNC:VCP)) increases tscript(complex(p(HGNC:NFKB1), p(HGNC:NFKB2), p(HGNC:REL),␣
↪p(HGNC:RELA), p(HGNC:RELB)))
tscript(complex(p(HGNC:NFKB1), p(HGNC:NFKB2), p(HGNC:REL), p(HGNC:RELA),␣
↪p(HGNC:RELB))) decreases bp(MESHPP:Apoptosis)

UNSET ALL
```

pybel.examples.**egf_graph**

This is the first attempt at curating an excerpt from the research article, "Genetics ignite focus on microglial inflammation in Alzheimer's disease".

```
SET Citation = {"PubMed", "26438529"}
SET Evidence = "Sialic acid binding activates CD33, resulting in phosphorylation of␣
↪the CD33 immunoreceptor tyrosine-based inhibitory motif (ITIM) domains and␣
↪activation of the SHP-1 and SHP-2 tyrosine phosphatases [66, 67]."
SET Species = 9606

complex(p(HGNC:CD33),a(CHEBI:"sialic acid")) -> p(HGNC:CD33, pmod(P))
act(p(HGNC:CD33, pmod(P))) => act(p(HGNC:PTPN6), ma(phos))
act(p(HGNC:CD33, pmod(P))) => act(p(HGNC:PTPN11), ma(phos))

UNSET {Evidence, Species}
SET Evidence = "These phosphatases act on multiple substrates, including Syk, to␣
↪inhibit immune activation [68, 69].  Hence, CD33 activation leads to increased SHP-
↪1 and SHP-2 activity that antagonizes Syk, inhibiting ITAM-signaling proteins,␣
↪possibly including TREM2/DAP12 (Fig. 1, [70, 71])."

act(p(HGNC:PTPN6)) =| act(p(HGNC:SYK))
act(p(HGNC:PTPN11)) =| act(p(HGNC:SYK))
act(p(HGNC:SYK)) -> act(p(HGNC:TREM2))
act(p(HGNC:SYK)) -> act(p(HGNC:TYROBP))

UNSET ALL
```

`pybel.examples.`**`sialic_acid_graph`**

## 2.5 Summary

`pybel.struct.summary.`**`get_syntax_errors`**`(`*graph*`)`
Gets a list of the syntax errors from the BEL script underlying the graph. Uses SyntaxError as a stand-in for `pybel.parser.parse_exceptions.BelSyntaxError`

**Parameters** **`graph`** (`pybel.BELGraph`) – A BEL graph

**Returns** A list of 4-tuples of line number, line text, exception, and annotations present in the parser

**Return type** list[tuple]

`pybel.struct.summary.`**`get_functions`**`(`*graph*`)`
Gets the set of all functions used in this graph

**Parameters** **`graph`** (`pybel.BELGraph`) – A BEL graph

**Returns** A set of functions

**Return type** set[str]

`pybel.struct.summary.`**`count_functions`**`(`*graph*`)`
Counts the frequency of each function present in a graph

**Parameters** **`graph`** (`pybel.BELGraph`) – A BEL graph

**Returns** A Counter from {function: frequency}

**Return type** collections.Counter

`pybel.struct.summary.`**`count_namespaces`**`(`*graph*`)`
Counts the frequency of each namespace across all nodes (that have namespaces)

**Parameters** **`graph`** (`pybel.BELGraph`) – A BEL graph

**Returns** A Counter from {namespace: frequency}

**Return type** collections.Counter

`pybel.struct.summary.`**`get_namespaces`**`(`*graph*`)`
Gets the set of all namespaces used in this graph

**Parameters** **`graph`** (`pybel.BELGraph`) – A BEL graph

**Returns** A set of namespaces

**Return type** set[str]

`pybel.struct.summary.`**`get_names_by_namespace`**`(`*graph*, *namespace*`)`
Get the set of all of the names in a given namespace that are in the graph. Raises `IndexError` if the namespace is not defined in the graph.

**Parameters**

- **`graph`** (`pybel.BELGraph`) – A BEL graph
- **`namespace`** (*`str`*) – A namespace keyword

**Returns** A set of names belonging to the given namespace that are in the given graph

**Return type** set[str]

pybel.struct.summary.**get_unused_namespaces**(*graph*)
Gets the set of all namespaces that are defined in a graph, but are never used.

**Parameters** **graph** (pybel.BELGraph) – A BEL graph

**Returns** A set of namespaces that are included but not used

**Return type** set[str]

pybel.struct.summary.**iterate_pubmed_identifiers**(*graph*)
Iterates over all PubMed identifiers in a graph

**Parameters** **graph** (pybel.BELGraph) – A BEL graph

**Returns** An iterator over the PubMed identifiers in the graph

**Return type** iter[str]

pybel.struct.summary.**get_pubmed_identifiers**(*graph*)
Gets the set of all PubMed identifiers cited in the construction of a graph

**Parameters** **graph** (pybel.BELGraph) – A BEL graph

**Returns** A set of all PubMed identifiers cited in the construction of this graph

**Return type** set[str]

pybel.struct.summary.**iter_annotation_value_pairs**(*graph*)
Iterates over the key/value pairs, with duplicates, for each annotation used in a BEL graph

**Parameters** **graph** (pybel.BELGraph) – A BEL graph

**Return type** iter[tuple[str,str]]

pybel.struct.summary.**iter_annotation_values**(*graph*, *annotation*)
Iterates over all of the values for an annotation used in the graph

**Parameters**

- **graph** (pybel.BELGraph) – A BEL graph
- **annotation** (*str*) – The annotation to grab

**Return type** iter[str]

pybel.struct.summary.**get_annotation_values_by_annotation**(*graph*)
Gets the set of values for each annotation used in a BEL graph

**Parameters** **graph** (pybel.BELGraph) – A BEL graph

**Returns** A dictionary of {annotation key: set of annotation values}

**Return type** dict[str, set[str]]

## 2.6 Filters

This module contains functions for filtering node and edge iterables. It relies heavily on the concepts of functional programming and the concept of predicates.

pybel.struct.filters.**concatenate_node_predicates**(*node_predicates=None*)
Concatenates multiple node filters to a new filter that requires all filters to be met

**Parameters** **node_predicates** (*types.FunctionType or iter[types.FunctionType]*) – A predicate or list of predicates (graph, node) -> bool

**Returns** A combine predicate (graph, node) -> bool

**Return type** types.FunctionType

Example usage:

```
>>> from pybel.dsl import protein, gene
>>> from pybel.struct.filters.node_predicates import not_pathology, node_
↪exclusion_predicate_builder
>>> app_protein = protein(name='APP', namespace='HGNC')
>>> app_gene = gene(name='APP', namespace='HGNC')
>>> app_filter = node_exclusion_predicate_builder([app_protein, app_gene])
>>> my_filter = concatenate_node_predicates([not_pathology, app_filter])
```

`pybel.struct.filters.`**`filter_nodes`**`(`*graph*, *node_predicates=None*`)`

Applies a set of predicates to the nodes iterator of a BEL graph

**Parameters**

- **graph** (`BELGraph`) – A BEL graph
- **node_predicates** (*types.FunctionType or iter[types.FunctionType]*) – A node predicate or list/tuple of node predicates

**Returns** An iterable of nodes that pass all predicates

**Return type** iter

`pybel.struct.filters.`**`get_nodes`**`(`*graph*, *node_predicates=None*`)`

Gets the set of all nodes that pass the predicates

**Parameters**

- **graph** (`BELGraph`) – A BEL graph
- **node_predicates** (*types.FunctionType or iter[types.FunctionType]*) – A node predicate or list/tuple of node predicates

**Returns** The set of nodes passing the predicates

**Return type** set[tuple]

`pybel.struct.filters.`**`count_passed_node_filter`**`(`*graph*, *node_predicates=None*`)`

Counts how many nodes pass a given set of filters

**Parameters**

- **graph** (`pybel.BELGraph`) – A BEL graph
- **node_predicates** (*types.FunctionType or iter[types.FunctionType]*) – A node predicate or list/tuple of node predicates

**Returns** The number of nodes passing the given set of predicates

**Return type** int

`pybel.struct.filters.`**`invert_edge_filter`**`(`*edge_predicate*`)`

Builds a filter that is the inverse of the given filter

**Parameters** **edge_predicate** (*types.FunctionType*) – An edge filter function (graph, node, node, key, data) -> bool

**Returns** An edge filter function

**Return type** (*pybel.BELGraph*, tuple, tuple, int) -> bool `pybel.struct.filters.`**`and_edge_predicates`**(*edge_predicates=None*)

Concatenates multiple edge predicates to a new predicate that requires all predicates to be met.

**Parameters** **edge_predicates** (*Optional[(pybel.BELGraph, tuple, tuple, int) -> bool or iter[(pybel.BELGraph, tuple, tuple, int) -> bool]]*) – a list of predicates (graph, node, node, key, data) -> bool

**Returns** A combine filter

**Return type** (*pybel.BELGraph*, tuple, tuple, int) -> bool

`pybel.struct.filters.`**`filter_edges`**(*graph*, *edge_predicates=None*)

Applies a set of filters to the edges iterator of a BEL graph

**Parameters**

- **graph** (`BELGraph`) – A BEL graph
- **edge_predicates** (*Optional[(pybel.BELGraph, tuple, tuple, int) -> bool or iter[(pybel.BELGraph, tuple, tuple, int) -> bool]]*) – A predicate or list of predicates

**Returns** An iterable of edges that pass all predicates

**Return type** iter[tuple,tuple,int]

`pybel.struct.filters.`**`count_passed_edge_filter`**(*graph*, *edge_predicates=None*)

Returns the number of edges passing a given set of predicates

**Parameters**

- **graph** (`pybel.BELGraph`) – A BEL graph
- **edge_predicates** (*Optional[(pybel.BELGraph, tuple, tuple, int) -> bool or iter[(pybel.BELGraph, tuple, tuple, int) -> bool]]*) – A predicate or list of predicates

**Returns** The number of edges passing a given set of predicates

**Return type** int

`pybel.struct.filters.`**`edge_predicate`**(*f*)

Apply this as a decorator to a function that takes a single argument, a PyBEL node data dictionary, to make sure that it can also accept a pair of arguments, a BELGraph and a PyBEL node tuple as well.

**Return type** (*pybel.BELGraph*, tuple, tuple, int) -> bool

`pybel.struct.filters.`**`has_pubmed`**(*data*)

Checks if the edge data dictionary has a PubMed citation

**Parameters** **data** (*dict*) – A PyBEL edge data dictionary from a *pybel.BELGraph*

**Returns** Does the edge data dictionary has a PubMed citation?

**Return type** bool

`pybel.struct.filters.`**`has_provenance`**(*data*)

Passes for edges with provenance information (i.e. citation and evidence)

**Parameters** **data** (*dict*) – The edge data dictionary

**Returns** If the edge has both a citation and and evidence entry

**Return type** bool

`pybel.struct.filters.`**`has_authors`**(*data*)

Passes for edges that have citations with authors

**Parameters** **data** (*dict*) – A PyBEL edge data dictionary from a *pybel.BELGraph*

**Returns** Does the edge's citation data dictionary have authors included?

**Return type** bool

`pybel.struct.filters.`**`is_causal_relation`**(*data*)
Is the given relation causal?

**Parameters** **data** (*dict*) – The PyBEL edge data dictionary

**Return type** bool

`pybel.struct.filters.`**`is_direct_causal_relation`**(*data*)
Checks if the edge is a direct causal relation

**Parameters** **data** (*dict*) – The PyBEL edge data dictionary

**Return type** bool

`pybel.struct.filters.`**`is_associative_relation`**(*data*)
Only passes on associative edges

**Parameters** **data** (*dict*) – The PyBEL edge data dictionary

**Returns** If the edge is a causal edge

**Return type** bool

`pybel.struct.filters.`**`has_polarity`**(*data*)
Only passes on polarized edges, belonging to the set *pybel.constants.CAUSAL_RELATIONS* or

**Parameters** **data** (*dict*) – The edge data dictionary

**Returns** If the edge is a polar edge

**Return type** bool

`pybel.struct.filters.`**`edge_has_activity`**(*data*)
Checks if the edge contains an activity in either the subject or object

**Parameters** **data** (*dict*) – The edge data dictionary

**Returns** If the edge contains an activity in either the subject or object

**Return type** bool

`pybel.struct.filters.`**`edge_has_degradation`**(*data*)
Checks if the edge contains a degradation in either the subject or object

**Parameters** **data** (*dict*) – The edge data dictionary

**Returns** If the edge contains a degradation in either the subject or object

**Return type** bool

`pybel.struct.filters.`**`edge_has_translocation`**(*data*)
Checks if the edge has a translocation in either the subject or object

**Parameters** **data** (*dict*) – The edge data dictionary

**Returns** If the edge has a translocation in either the subject or object

**Return type** bool

`pybel.struct.filters.`**`edge_has_annotation`**(*data*, *key*)
Checks that ANNOTATION is included in the data dictionary and that the key is also present

**Parameters**

- **data** (*dict*) – The data dictionary from a BELGraph's edge
- **key** (*str*) – An annotation key

**Returns** If the annotation key is present in the current data dictionary

**Return type** Optional[Any]

For example, it might be useful to print all edges that are annotated with 'Subgraph':

```
>>> from pybel.examples import sialic_acid_graph
>>> for u, v, data in sialic_acid_graph.edges_iter(data=True):
>>>     if edge_has_annotation(data, 'Species')
>>>         print(u, v, data)
```

`pybel.struct.filters.`**`keep_node_permissive`**`(`*graph*, *node*`)`

A default node predicate that always evaluates to `True`.

Given BEL graph `graph`, applying *`keep_node_permissive()`* with a predicate on the nodes iterable as in `filter(keep_node_permissive, graph)` will result in the same iterable as iterating directly over a `BELGraph`

**Parameters**

- **graph** (`BELGraph`) – A BEL graph
- **node** (*tuple*) – The node

**Returns** Always returns `True`

**Return type** bool

`pybel.struct.filters.`**`is_abundance`**`(`*data*`)`

Returns true if the node is an abundance

**Parameters** **data** (*dict*) – A PyBEL data dictionary

**Return type** bool

`pybel.struct.filters.`**`is_gene`**`(`*data*`)`

Returns true if the node is a gene

**Parameters** **data** (*dict*) – A PyBEL data dictionary

**Return type** bool

`pybel.struct.filters.`**`is_protein`**`(`*data*`)`

Returns true if the node is a protein

**Parameters** **data** (*dict*) – A PyBEL data dictionary

**Return type** bool

`pybel.struct.filters.`**`is_pathology`**`(`*data*`)`

Returns true if the node is a pathology

**Parameters** **data** (*dict*) – A PyBEL data dictionary

**Return type** bool

`pybel.struct.filters.`**`not_pathology`**`(`*data*`)`

Returns false if the node is a pathology

**Parameters** **data** (*dict*) – A PyBEL data dictionary

**Return type** bool

`pybel.struct.filters.`**`has_variant`**`(`*data*`)`
Returns true if the node has any variants

**Parameters** **`data`** (*`dict`*) – A PyBEL data dictionary

**Return type** bool

`pybel.struct.filters.`**`has_protein_modification`**`(`*data*`)`
Returns true if the node has a protein modification variant

**Parameters** **`data`** (*`dict`*) – A PyBEL data dictionary

**Return type** bool

`pybel.struct.filters.`**`has_gene_modification`**`(`*data*`)`
Checks if a node has a gene modification

**Parameters** **`data`** (*`dict`*) – A PyBEL data dictionary

**Return type** bool

`pybel.struct.filters.`**`has_hgvs`**`(`*data*`)`
Checks if a node has an HGVS variant

**Parameters** **`data`** (*`dict`*) – A PyBEL data dictionary

**Return type** bool

`pybel.struct.filters.`**`has_fragment`**`(`*data*`)`
Checks if a node has a fragment

**Parameters** **`data`** (*`dict`*) – A PyBEL data dictionary

**Return type** bool

`pybel.struct.filters.`**`has_activity`**`(`*graph*, *node*`)`
Returns true if over any of the node's edges it has a molecular activity

**Parameters**

- **`graph`** (`pybel.BELGraph`) – A BEL graph
- **`node`** (*`tuple`*) – A BEL node

**Returns** If the node has a known molecular activity

**Return type** bool

`pybel.struct.filters.`**`is_degraded`**`(`*graph*, *node*`)`
Returns true if over any of the node's edges it is degraded

**Parameters**

- **`graph`** (`pybel.BELGraph`) – A BEL graph
- **`node`** (*`tuple`*) – A BEL node

**Returns** If the node has a known degradation

**Return type** bool

`pybel.struct.filters.`**`is_translocated`**`(`*graph*, *node*`)`
Returns true if over any of the node's edges it is transloated

**Parameters**

- **`graph`** (`pybel.BELGraph`) – A BEL graph
- **`node`** (*`tuple`*) – A BEL node

**Returns** If the node has a known translocation

**Return type** bool

`pybel.struct.filters.`**`has_causal_in_edges`**(*graph*, *node*)
Returns true if the node contains any in_edges that are causal

**Parameters**

- **`graph`** (`pybel.BELGraph`) – A BEL graph
- **`node`** (*`tuple`*) – A BEL node

**Return type** bool

`pybel.struct.filters.`**`has_causal_out_edges`**(*graph*, *node*)
Returns true if the node contains any out_edges that are causal

**Parameters**

- **`graph`** (`pybel.BELGraph`) – A BEL graph
- **`node`** (*`tuple`*) – A BEL node

**Return type** bool

`pybel.struct.filters.`**`node_inclusion_predicate_builder`**(*nodes*)
Builds a function that returns true

**Parameters or dict] nodes** (*`list[tuple`*) – A list of PyBEL node data dictionaries or PyBEL node tuples

**Return type** types.FunctionType

`pybel.struct.filters.`**`node_exclusion_predicate_builder`**(*nodes*)
Builds a function that returns true

**Parameters or dict] nodes** (*`list[tuple`*) – A list of PyBEL node data dictionaries or PyBEL node tuples

**Return type** types.FunctionType

`pybel.struct.filters.`**`is_causal_source`**(*graph*, *node*)
Is the node is a causal source?

- Doesn't have any causal in edge(s)
- Does have causal out edge(s)

**Parameters**

- **`graph`** (`pybel.BELGraph`) – A BEL graph
- **`node`** (*`tuple`*) – A BEL node

**Returns** If the node is a causal source

**Return type** bool

`pybel.struct.filters.`**`is_causal_sink`**(*graph*, *node*)
Is the node is a causal sink?

- Does have causal in edge(s)
- Doesn't have any causal out edge(s)

**Parameters**

- **graph** (pybel.BELGraph) – A BEL graph
- **node** (*tuple*) – A BEL node

**Returns** If the node is a causal source

**Return type** bool

`pybel.struct.filters.`**`is_causal_central`**(*graph*, *node*)
Is the node neither a causal sink nor a causal source?

- Does have causal in edges(s)
- Does have causal out edge(s)

**Parameters**

- **graph** (pybel.BELGraph) – A BEL graph
- **node** (*tuple*) – A BEL node

**Returns** If the node is neither a causal sink nor a causal source

**Return type** bool

`pybel.struct.filters.`**`build_annotation_dict_all_filter`**(*annotations*)
Builds a filter that keeps edges whose data dictionaries's annotations entry are super-dictionaries to the given dictionary

**Parameters** **annotations** (*dict*) – The annotation query dict to match

**Return type** (*pybel.BELGraph*, tuple, tuple, int) -> bool

`pybel.struct.filters.`**`build_annotation_dict_any_filter`**(*annotations*)
Builds a filter that keeps edges whose data dictionaries's annotations entry contain any match to the target dictionary

**Parameters** **annotations** (*dict*) – The annotation query dict to match

**Return type** (*pybel.BELGraph*, tuple, tuple, int) -> bool

`pybel.struct.filters.`**`part_has_modifier`**(*data*, *part*, *modifier*)
Returns true if the modifier is in the given subject/object part

**Parameters**

- **data** (*dict*) – A PyBEL edge data dictionary
- **part** (*str*) – either *pybel.constants.SUBJECT* or *pybel.constants.OBJECT*
- **modifier** – The modifier to look for

**Return type** bool

## 2.7 Mutation

This module contains functions that mutate or make transformations on a network

`pybel.struct.mutation.`**`strip_annotations`**(*graph*)
Strips all the annotations from a BEL graph

**Parameters** **graph** (pybel.BELGraph) – A BEL graph `pybel.struct.mutation.`**`infer_child_relations`**`(`*graph*, *node*`)`

Propagates causal relations to children

**Parameters**

- **`graph`** (`pybel.BELGraph`) – A BEL graph
- **`node`** (*`tuple or BaseEntity`*) – A PyBEL node tuple, on which to propagate the children's relations

## 2.8 Input and Output

PyBEL provides multiple lossless interchange options for BEL. Lossy output formats are also included for convenient export to other programs. Notably, a *de facto* interchange using Resource Description Framework (RDF) to match the ability of other existing software is excluded due the immaturity of the BEL to RDF mapping.

### 2.8.1 Import

#### Parsing Modes

The PyBEL parser has several modes that can be enabled and disabled. They are described below.

#### Allow Naked Names

By default, this is set to `False`. The parser does not allow identifiers that are not qualified with namespaces (*naked names*), like in `p(YFG)`. A proper namespace, like `p(HGNC:YFG)` must be used. By setting this to `True`, the parser becomes permissive to naked names. In general, this is bad practice and this feature will be removed in the future.

#### Allow Nested

By default, this is set to `False`. The parser does not allow nested statements is disabled. See *overview*. By setting this to `True` the parser will accept nested statements one level deep.

#### Citation Clearing

By default, this is set to `True`. While the BEL specification clearly states how the language should be used as a state machine, many BEL documents do not conform to the strict `SET`/`UNSET` rules. To guard against annotations accidentally carried from one set of statements to the next, the parser has two modes. By default, in citation clearing mode, when a `SET CITATION` command is reached, it will clear all other annotations (except the `STATEMENT_GROUP`, which has higher priority). This behavior can be disabled by setting this to `False` to re-enable strict parsing.

#### Reference

`pybel.`**`from_lines`**`(`*lines*, *manager=None*, *allow_nested=False*, *citation_clearing=True*, ***kwargs*`)`

Loads a BEL graph from an iterable over the lines of a BEL script

**Parameters**

- **`lines`** (*`iter[str]`*) – An iterable of strings (the lines in a BEL script)

- **manager** (*Optional[str or pybel.manager.Manager]*) – database connection string to cache, pre-built `Manager`, or None to use default cache
- **citation_clearing** (*bool*) – Should `SET Citation` statements clear evidence and all annotations? Delegated to `pybel.parser.ControlParser`
- **kwargs** (*dict*) – keyword arguments to `pybel.io.line_utils.parse_lines()`

**Return type** *BELGraph*

`pybel.`**`from_path`**(*path*, *manager=None*, *allow_nested=False*, *citation_clearing=True*, *encoding='utf-8'*, ***kwargs*)

Loads a BEL graph from a file resource. This function is a thin wrapper around `from_lines()`.

**Parameters**

- **path** (*str*) – A file path
- **manager** (*None or str or pybel.manager.Manager*) – database connection string to cache, pre-built `Manager`, or None to use default cache
- **allow_nested** (*bool*) – if true, turn off nested statement failures
- **citation_clearing** (*bool*) – Should `SET Citation` statements clear evidence and all annotations? Delegated to `pybel.parser.ControlParser`
- **encoding** (*str*) – the encoding to use when reading this file. Is passed to `codecs.open`. See the python docs for a list of standard encodings. For example, files starting with a UTF-8 BOM should use `utf_8_sig`
- **kwargs** (*dict*) – keyword arguments to `pybel.io.line_utils.parse_lines()`

**Return type** *BELGraph*

`pybel.`**`from_url`**(*url*, *manager=None*, *allow_nested=False*, *citation_clearing=True*, ***kwargs*)

Loads a BEL graph from a URL resource. This function is a thin wrapper around `from_lines()`.

**Parameters**

- **url** (*str*) – A valid URL pointing to a BEL resource
- **manager** (*None or str or pybel.manager.Manager*) – database connection string to cache, pre-built `Manager`, or None to use default cache
- **allow_nested** (*bool*) – if true, turn off nested statement failures
- **citation_clearing** (*bool*) – Should `SET Citation` statements clear evidence and all annotations? Delegated to `pybel.parser.ControlParser`
- **kwargs** (*dict*) – keyword arguments to `pybel.io.line_utils.parse_lines()`

**Return type** *BELGraph*

### 2.8.2 Canonicalization

`pybel.`**`to_bel_lines`**(*graph*)

Returns an iterable over the lines of the BEL graph as a canonical BEL Script (.bel)

**Parameters** **graph** (`pybel.BELGraph`) – the BEL Graph to output as a BEL Script

**Returns** An iterable over the lines of the representative BEL script

**Return type** iter[str]

`pybel.`**`to_bel`**(*graph*, *file=None*)
Outputs the BEL graph as canonical BEL to the given file/file-like/stream. Defaults to standard out.

**Parameters**

- **graph** (`BELGraph`) – the BEL Graph to output as a BEL Script
- **file** (*file*) – A writable file-like object. If None, defaults to standard out.

`pybel.`**`to_bel_path`**(*graph*, *path*, *mode='w'*, ***kwargs*)
Writes the BEL graph as a canonical BEL Script to the given path

**Parameters**

- **graph** (`BELGraph`) – the BEL Graph to output as a BEL Script
- **path** (*str*) – A file path
- **mode** (*str*) – The file opening mode. Defaults to 'w'

### 2.8.3 Transport

All transport pairs are reflective and data-preserving.

#### Bytes

This module contains IO functions for interconversion between BEL graphs and python pickle objects

`pybel.`**`from_pickle`**(*path*, *check_version=True*)
Reads a graph from a gpickle file.

**Parameters**

- **path** (*str or file*) – File or filename to read. Filenames ending in .gz or .bz2 will be uncompressed.
- **check_version** (*bool*) – Checks if the graph was produced by this version of PyBEL

**Returns** A BEL graph

**Return type** *BELGraph*

`pybel.`**`to_pickle`**(*graph*, *file*, *protocol=4*)
Writes this graph to a pickle object with `networkx.write_gpickle()`. Note that the pickle module has some incompatibilities between Python 2 and 3. To export a universally importable pickle, choose 0, 1, or 2.

**Parameters**

- **graph** (`BELGraph`) – A BEL graph
- **file** (*str or file*) – A file or filename to write to
- **protocol** (*int*) – Pickling protocol to use

**See also:**

https://docs.python.org/3.6/library/pickle.html#data-stream-format

`pybel.`**`from_bytes`**(*bytes_graph*, *check_version=True*)
Reads a graph from bytes (the result of pickling the graph).

**Parameters**

- **bytes_graph** (*bytes*) – File or filename to write

- **check_version** (*bool*) – Checks if the graph was produced by this version of PyBEL

**Returns** A BEL graph

**Return type** *BELGraph*

`pybel.`**`to_bytes`**(*graph*, *protocol=4*)

Converts a graph to bytes with pickle. Note that the pickle module has some incompatibilities between Python 2 and 3. To export a universally importable pickle, choose 0, 1, or 2.

**Parameters**

- **graph** (`BELGraph`) – A BEL network
- **protocol** (*int*) – Pickling protocol to use

**Returns** Pickled bytes representing the graph

**Return type** bytes

**See also:**

https://docs.python.org/3.6/library/pickle.html#data-stream-format

## Node-Link JSON

This module contains IO functions for interconversion between BEL graphs and Node-Link JSON

`pybel.`**`from_json`**(*graph_json_dict*, *check_version=True*)

Builds a graph from Node-Link JSON Object

**Parameters**

- **graph_json_dict** (*dict*) – A JSON dictionary representing a graph
- **check_version** (*bool*) – Checks if the graph was produced by this version of PyBEL

**Return type** *BELGraph*

`pybel.`**`to_json`**(*graph*)

Converts this graph to a Node-Link JSON object

**Parameters** **graph** (`BELGraph`) – A BEL graph

**Returns** A Node-Link JSON object representing the given graph

**Return type** dict

`pybel.`**`from_json_file`**(*file*, *check_version=True*)

Builds a graph from the Node-Link JSON contained in the given file

**Parameters**

- **file** (*file*) – A readable file or file-like
- **check_version** (*bool*) – Checks if the graph was produced by this version of PyBEL

**Return type** *BELGraph*

`pybel.`**`to_json_file`**(*graph*, *file*, ***kwargs*)

Writes this graph as Node-Link JSON to a file

**Parameters**

- **graph** (`BELGraph`) – A BEL graph
- **file** (*file*) – A write-supporting file or file-like object

`pybel.`**`from_json_path`**(*path*, *check_version=True*)
Builds a graph from a file containing Node-Link JSON

**Parameters**

- **`path`** (*`str`*) – A file path. Expands user.
- **`check_version`** (*`bool`*) – Checks if the graph was produced by this version of PyBEL

**Return type** *BELGraph*

`pybel.`**`to_json_path`**(*graph*, *path*, ***kwargs*)
Writes this graph to the given path as a Node-Link JSON

**Parameters**

- **`graph`** (`BELGraph`) – A BEL graph
- **`path`** (*`str`*) – A file path

`pybel.`**`from_jsons`**(*graph_json_str*, *check_version=True*)
Reads a BEL graph from a Node-Link JSON string

**Parameters**

- **`graph_json_str`** (*`str`*) – A Node-Link JSON string produced by PyBEL
- **`check_version`** (*`bool`*) – Checks if the graph was produced by this version of PyBEL

**Return type** *BELGraph*

`pybel.`**`to_jsons`**(*graph*, ***kwargs*)
Dumps this graph as a Node-Link JSON object to a string

**Parameters** **`graph`** (`BELGraph`) – A BEL graph

**Returns** A string representation of the Node-Link JSON produced for this graph by *`pybel.to_json()`*

**Return type** str

## JSON Graph Interchange Format

The JSON Graph Interchange Format (JGIF) is specified similarly to the Node-Link JSON. Interchange with this format provides compatibilty with other software and repositories, such as the Causal Biological Network Database.

`pybel.`**`from_jgif`**(*graph_jgif_dict*)
Builds a BEL graph from a JGIF JSON object.

**Parameters** **`graph_jgif_dict`** (*`dict`*) – The JSON object representing the graph in JGIF format

**Return type** *BELGraph*

`pybel.`**`from_cbn_jgif`**(*graph_jgif_dict*)
Maps the JGIF used by the Causal Biological Network Database to standard namespace and annotations, then builds a BEL graph using *`pybel.from_jgif()`*.

**Parameters** **`graph_jgif_dict`** (*`dict`*) – The JSON object representing the graph in JGIF format

**Return type** *BELGraph*

Example:

```
>>> import requests
>>> from pybel import from_cbn_jgif
>>> apoptosis_url = 'http://causalbionet.com/Networks/GetJSONGraphFile?
↪networkId=810385422'
>>> graph_jgif_dict = requests.get(apoptosis_url).json()
>>> graph = from_cbn_jgif(graph_jgif_dict)
```

**Warning:** Handling the annotations is not yet supported, since the CBN documents do not refer to the resources used to create them. This may be added in the future, but the annotations must be stripped from the graph before uploading to the network store using *pybel.struct.mutation.strip_annotations()*

`pybel.`**`to_jgif`**`(`*graph*`)`

Builds a JGIF dictionary from a BEL graph.

**Parameters** **graph** (`pybel.BELGraph`) – A BEL graph

**Returns** A JGIF dictionary

**Return type** dict

**Warning:** Untested! This format is not general purpose and is therefore time is not heavily invested. If you want to use Cytoscape.js, we suggest using *pybel.to_cx()* instead.

Example:

```
>>> import pybel, os, json
>>> graph_url = 'https://arty.scai.fraunhofer.de/artifactory/bel/knowledge/
↪selventa-small-corpus/selventa-small-corpus-20150611.bel'
>>> graph = pybel.from_url(graph_url)
>>> graph_jgif_json = pybel.to_jgif(graph)
>>> with open(os.path.expanduser('~/Desktop/small_corpus.json'), 'w') as f:
...     json.dump(graph_jgif_json, f)
```

### CX JSON

CX is an aspect-oriented network interchange format encoded in JSON with a format inspired by the JSON-LD encoding of Resource Description Framework (RDF). It is primarily used by the Network Data Exchange (NDEx) and more recent versions of Cytoscape.

**See also:**

- The NDEx Data Model Specification
- Cytoscape.js
- CX Support for Cytoscape.js on the Cytoscape App Store

`pybel.`**`from_cx`**`(`*cx*`)`

Rebuilds a BELGraph from CX JSON output from PyBEL

**Parameters** **cx** (*list[dict]*) – The CX JSON for this graph

**Return type** *BELGraph*

`pybel.to_cx(graph)`

Converts BEL Graph to CX data format (as in-memory JSON) for use with NDEx

**Parameters** **graph** (`BELGraph`) – A BEL Graph

**Returns** The CX JSON for this graph

**Return type** list

**See also:**

- NDEx Python Client
- PyBEL / NDEx Python Client Wrapper

`pybel.from_cx_file(file)`

Reads a file containing CX JSON and converts to a BEL graph

**Parameters** **file** (`file`) – A readable file or file-like containing the CX JSON for this graph

**Returns** A BEL Graph representing the CX graph contained in the file

**Return type** *BELGraph*

`pybel.to_cx_file(graph, file, indent=2, **kwargs)`

Writes this graph to a JSON file in CX format

**Parameters**

- **graph** (`BELGraph`) – A BEL graph
- **file** (`file`) – A writable file or file-like
- **indent** (`Optional[int]`) – How many spaces to use to pretty print. Change to None for no pretty printing

Example:

```
>>> from pybel import from_url, to_cx_file
>>> from pybel.constants import SMALL_CORPUS_URL
>>> graph = from_url(SMALL_CORPUS_URL)
>>> with open('graph.cx', 'w') as f:
>>> ... to_cx_file(graph, f)
```

`pybel.from_cx_jsons(graph_cx_json_str)`

Reconstitutes a BEL graph from a CX JSON string

**Parameters** **graph_cx_json_str** (`str`) – CX JSON string

**Returns** A BEL graph representing the CX graph contained in the string

**Return type** *BELGraph*

`pybel.to_cx_jsons(graph, **kwargs)`

Dumps a BEL graph as CX JSON to a string

**Parameters** **graph** (`BELGraph`) – A BEL Graph

**Returns** CX JSON string

**Return type** str

### 2.8.4 Export

`pybel.`**`to_graphml`**(*graph*, *file*)

Writes this graph to GraphML XML file using `networkx.write_graphml()`. The .graphml file extension is suggested so Cytoscape can recognize it.

**Parameters**

- **graph** (`BELGraph`) – A BEL graph
- **file** (*file*) – A file or file-like object

`pybel.`**`to_csv`**(*graph*, *file=None*, *sep='\t'*)

Writes the graph as a tab-separated edge list with the columns:

1. Source BEL term
2. Relation
3. Target BEL term
4. Edge data dictionary.

See the Data Models section of the documentation for which data are stored in the edge data dictionary, such as queryable information about transforms on the subject and object and their associated metadata.

**Parameters**

- **graph** (`BELGraph`) – A BEL graph
- **file** (*file*) – A writable file or file-like. Defaults to stdout.
- **sep** (*str*) – The separator. Defaults to tab.

`pybel.`**`to_sif`**(*graph*, *file=None*, *sep='\t'*)

Writes the graph as a tab-separated SIF file with the following columns:

1. Source BEL term
2. Relation
3. Target BEL term

This format is simple and can be used readily with many applications, but is lossy in that it does not include relation metadata.

**Parameters**

- **graph** (`BELGraph`) – A BEL graph
- **file** (*file*) – A writable file or file-like. Defaults to stdout.
- **sep** (*str*) – The separator. Defaults to tab.

`pybel.`**`to_gsea`**(*graph*, *file=None*)

Writes the genes/gene products to a GRP file for use with GSEA gene set enrichment analysis

**Parameters**

- **graph** (`BELGraph`) – A BEL graph
- **file** (*file*) – A writeable file or file-like object. Defaults to stdout.

**See also:**

- GRP format specification
- GSEA publication

## 2.8.5 Database

### SQL Database

This module provides IO functions to the relational edge store.

`pybel.`**`from_database`**(*name*, *version=None*, *connection=None*)

Loads a BEL graph from a database. If name and version are given, finds it exactly with `pybel.manager.Manager.get_network_by_name_version()`. If just the name is given, finds most recent with `pybel.manager.Manager.get_network_by_name_version()`

**Parameters**

- **`name`** (*`str`*) – The name of the graph
- **`version`** (*`str`*) – The version string of the graph. If not specified, loads most recent graph added with this name
- **`connection`** (*`None or str or pybel.manager.Manager`*) – An RFC-1738 database connection string, a pre-built `Manager`, or `None` for default connection

**Returns** A BEL graph loaded from the database

**Return type** Optional[*BELGraph*]

`pybel.`**`to_database`**(*graph*, *connection=None*, *store_parts=True*)

Stores a graph in a database.

**Parameters**

- **`graph`** (`BELGraph`) – A BEL graph
- **`connection`** (*`None or str or pybel.manager.Manager`*) – An RFC-1738 database connection string, a pre-built `Manager`, or *None*‘ for default connection
- **`store_parts`** (*`bool`*) – Should the graph be stored in the edge store?

**Returns** If successful, returns the network object from the database.

**Return type** Optional[*Network*]

### Neo4j

This module contains IO functions for outputting BEL graphs to a Neo4J graph database

`pybel.`**`to_neo4j`**(*graph*, *neo_connection*, *context=None*)

Uploads a BEL graph to Neo4J graph database using `py2neo`

**Parameters**

- **`graph`** (`BELGraph`) – A BEL Graph
- **`neo_connection`** (`py2neo.Graph`) – A `py2neo` connection object. Refer to the py2neo documentation for how to build this object.
- **`context`** (*`str`*) – A disease context to allow for multiple disease models in one neo4j instance. Each edge will be assigned an attribute `pybel_context` with this value

Example Usage:

```
>>> import pybel, py2neo
>>> url = 'http://resource.belframework.org/belframework/1.0/knowledge/small_
↪corpus.bel'
>>> g = pybel.from_url(url)
>>> neo_graph = py2neo.Graph("http://localhost:7474/db/data/")  # use your own␣
↪connection settings
>>> pybel.to_neo4j(g, neo_graph)
```

### Network Data Exchange (NDEx)

This package provides a wrapper around `pybel.to_cx()` and NDEx client to allow for easy upload and download of BEL documents to the NDEx database.

The programmatic API also provides options for specifying username and password. By default, it checks the environment variables: `NDEX_USERNAME` and `NDEX_PASSWORD`.

`pybel.`**`from_ndex`**(*network_id*, *username=None*, *password=None*, *debug=False*)
Downloads a BEL Graph from NDEx

**Warning:** This function only will work for CX documents that have been originally exported from PyBEL

**Parameters**

- **network_id** (*str*) – The UUID assigned to the network by NDEx
- **username** (*str*) – NDEx username
- **password** (*str*) – NDEx password
- **debug** (*bool*) – If true, turn on NDEx client debugging

**Returns** A BEL graph

**Return type** *BELGraph*

Example Usage:

```
>>> from pybel import from_ndex
>>> network_id = '1709e6f3-04a1-11e7-aba2-0ac135e8bacf'
>>> graph = from_ndex(network_id)
```

`pybel.`**`to_ndex`**(*graph*, *username=None*, *password=None*, *debug=False*)
Uploads a BEL graph to NDEx

**Parameters**

- **graph** (`BELGraph`) – A BEL graph
- **username** (*str*) – NDEx username
- **password** (*str*) – NDEx password
- **debug** (*bool*) – If true, turn on NDEx client debugging

**Returns** The UUID assigned to the network by NDEx

**Return type** str

Example Usage:

```
>>> import pybel
>>> graph = pybel.from_url('http://resources.openbel.org/belframework/20150611/
↪knowledge/small_corpus.bel')
>>> pybel.to_ndex(graph)
```

### 2.8.6 PyBEL Web

This module facilitates rudimentary data exchange with PyBEL Web.

**Warning:** These functions are hard to unit test because they rely on a web service that isn't *exactly* stable yet. Stay tuned!

`pybel.io.web.`**`to_web`**(*graph*, *host=None*)

Sends a graph to the receiver service and returns the `requests` response object

**Parameters**

- **`graph`** (`pybel.BELGraph`) – A BEL network
- **`host`** (*`Optional[str]`*) – The location of the PyBEL web server. Defaults to *`pybel.constants.DEFAULT_SERVICE_URL`*

**Returns** The response object from `requests`

**Return type** requests.Response

`pybel.io.web.`**`from_web`**(*network_id*, *host=None*)

Retrieves a public network from PyBEL Web. In the future, this function may be extended to support authentication.

**Parameters**

- **`network_id`** (*`int`*) – The PyBEL Web network identifier
- **`host`** (*`Optional[str]`*) – The location of the PyBEL web server. Defaults to *`pybel.constants.DEFAULT_SERVICE_URL`*

**Return type** *pybel.BELGraph*

### 2.8.7 INDRA

After assembling a model with INDRA, a list of `indra.statements.Statement` can be converted to a *`pybel.BELGraph`* with `indra.assemblers.PybelAssembler`.

```
from indra.assemblers import PybelAssembler
import pybel

stmts = [
    # A list of INDRA statements
]

pba = PybelAssembler(
    stmts,
    name='Graph Name',
    version='0.0.1',
    description='Graph Description'
)
graph = pba.make_model()

# Write to BEL file
pybel.to_bel_path(belgraph, 'simple_pybel.bel')
```

**Warning:** These functions are hard to unit test because they rely on a whole set of java dependencies and will likely not be for a while.

`pybel.io.indra.`**`from_indra_statements`**(*statements*, *name=None*, *version=None*, *description=None*)

Imports a model from `indra`.

**Parameters**

- **`statements`** (*`list[indra.statements.Statement]`*) – A list of statements
- **`name`** (*`str`*) – The name for the BEL graph
- **`version`** (*`str`*) – The version of the BEL graph
- **`description`** (*`str`*) – The description of the BEL graph

**Return type** *pybel.BELGraph*

`pybel.io.indra.`**`from_indra_pickle`**(*path*, *name=None*, *version=None*, *description=None*)

Imports a model from `indra`.

**Parameters**

- **`path`** (*`str`*) – Path to pickled list of `indra.statements.Statement`
- **`name`** (*`str`*) – The name for the BEL graph
- **`version`** (*`str`*) – The version of the BEL graph
- **`description`** (*`str`*) – The description of the BEL graph

**Return type** *pybel.BELGraph*

`pybel.io.indra.`**`to_indra`**(*graph*)

Exports this graph as a list of INDRA statements using *indra.sources.pybel.PybelProcessor*

**Parameters** **`graph`** (`pybel.BELGraph`) – A BEL graph

**Return type** list[indra.statements.Statement]

**Warning:** Not fully implemented yet! Needs the pybel_client branch of sorgerlab/indra

`pybel.io.indra.`**`from_biopax`**(*path*, *name=None*, *version=None*, *description=None*)

Imports a model encoded in BioPAX via `indra`.

**Parameters**

- **`path`** (*`str`*) – Path to a BioPAX OWL file
- **`name`** (*`str`*) – The name for the BEL graph
- **`version`** (*`str`*) – The version of the BEL graph
- **`description`** (*`str`*) – The description of the BEL graph

**Return type** *pybel.BELGraph*

## 2.9 Manager

### 2.9.1 Manager API

The BaseManager takes care of building and maintaining the connection to the database via SQLAlchemy.

**class** `pybel.manager.`**`BaseManager`**(*connection=None*, *echo=False*, *autoflush=None*, *autocommit=None*, *expire_on_commit=None*, *scopefunc=None*)

Creates a connection to database and a persistent session using SQLAlchemy

A custom default can be set as an environment variable with the name `pybel.constants.PYBEL_CONNECTION`, using an RFC-1738 string. For example, a MySQL string can be given with the following form:

`mysql+pymysql://<username>:<password>@<host>/<dbname>?charset=utf8[&<options>]`

A SQLite connection string can be given in the form:

`sqlite:///~/Desktop/cache.db`

Further options and examples can be found on the SQLAlchemy documentation on engine configuration.

**Parameters**

- **`connection`** (`str`) – An RFC-1738 database connection string. If `None`, tries to load from the environment variable `PYBEL_CONNECTION` then from the config file `~/.config/pybel/config.json` whose value for `PYBEL_CONNECTION` defaults to `pybel.constants.DEFAULT_CACHE_LOCATION`
- **`echo`** (`bool`) – Turn on echoing sql
- **`autoflush`** (`bool`) – Defaults to True if not specified in kwargs or configuration.
- **`autocommit`** (`bool`) – Defaults to False if not specified in kwargs or configuration.
- **`expire_on_commit`** (`bool`) – Defaults to False if not specified in kwargs or configuration.
- **`scopefunc`** – Scoped function to pass to `sqlalchemy.orm.scoped_session()`

From the Flask-SQLAlchemy documentation:

An extra key `'scopefunc'` can be set on the `options` dict to specify a custom scope function. If it's not provided, Flask's app context stack identity is used. This will ensure that sessions are created and removed with the request/response cycle, and should be fine in most cases.

**`session_maker = None`**

A SQLAlchemy session maker

**`session = None`**

A SQLAlchemy session object

**`create_all`**(*checkfirst=True*)

Creates the PyBEL cache's database and tables

**Parameters** **`checkfirst`** (`bool`) – Check if the database is made before trying to re-make it

**`drop_all`**(*checkfirst=True*)

Drops all data, tables, and databases for the PyBEL cache

The Manager collates multiple groups of functions for interacting with the database. For sake of code clarity, they are separated across multiple classes that are documented below.

**class** pybel.manager.**Manager**(**args*, ***kwargs*)

Bases: pybel.manager.query_manager.QueryManager, pybel.manager.cache_manager.InsertManager, pybel.manager.cache_manager.NetworkManager, pybel.manager.cache_manager.EquivalenceManager, pybel.manager.cache_manager.OwlNamespaceManager, pybel.manager.cache_manager.OwlAnnotationManager

The definition cache manager takes care of storing BEL namespace and annotation files for later use. It uses SQLite by default for speed and lightness, but any database can be used with its SQLAlchemy interface.

**static ensure**(*connection=None*, **args*, ***kwargs*)

A convenience method for turning a string into a connection, or passing a *Manager* through.

**Parameters**

- **connection** (*Optional[str or Manager]*) – An RFC-1738 database connection string, a pre-built *Manager*, or None for default connection
- **args** (*list*) – Positional arguments to pass to the constructor of *Manager*
- **kwargs** (*dict*) – Keyword arguments to pass to the constructor of *Manager*

**Return type** *Manager*

## 2.9.2 Manager Components

**class** pybel.manager.**NetworkManager**(*use_namespace_cache=False*, **args*, ***kwargs*)

Groups functions for inserting and querying networks in the database's network store.

**Parameters** **use_namespace_cache** – Should namespaces be cached in-memory?

**count_networks**()

Counts the number of networks in the cache

**Return type** int

**list_networks**()

Lists all networks in the cache

**Return type** list[*Network*]

**list_recent_networks**()

Lists the most recently created version of each network (by name)

**Return type** list[*Network*]

**has_name_version**(*name*, *version*)

Checks if the name/version combination is already in the database

**Parameters**

- **name** (*str*) – The network name
- **version** (*str*) – The network version

**Return type** bool

**static iterate_singleton_edges_from_network**(*network*)

Gets all edges that only belong to the given network

**Return type** iter[*Edge*]

**drop_network**(*network*)
Drops a network, while also cleaning up any edges that are no longer part of any network.

**drop_network_by_id**(*network_id*)
Drops a network by its database identifier

**Parameters** **network_id** (*int*) – The network's database identifier

**drop_networks**()
Drops all networks

**get_network_versions**(*name*)
Returns all of the versions of a network with the given name

**Parameters** **name** (*str*) – The name of the network to query

**Return type** set[str]

**get_network_by_name_version**(*name*, *version*)
Loads most network with the given name and version

**Parameters**

- **name** (*str*) – The name of the network.
- **version** (*str*) – The version string of the network.

**Return type** Optional[*Network*]

**get_graph_by_name_version**(*name*, *version*)
Loads most recently added graph with the given name, or allows for specification of version

**Parameters**

- **name** (*str*) – The name of the network.
- **version** (*str*) – The version string of the network.

**Return type** Optional[*BELGraph*]

**get_networks_by_name**(*name*)
Gets all networks with the given name. Useful for getting all versions of a given network.

**Parameters** **name** (*str*) – The name of the network

**Return type** list[*Network*]

**get_most_recent_network_by_name**(*name*)
Gets the most recently created network with the given name.

**Parameters** **name** (*str*) – The name of the network

**Return type** Optional[*Network*]

**get_graph_by_most_recent**(*name*)
Gets the most recently created network with the given name as a `pybel.BELGraph`.

**Parameters** **name** (*str*) – The name of the network

**Return type** Optional[*BELGraph*]

**get_network_by_id**(*network_id*)
Gets a network from the database by its identifier.

**Parameters** **network_id** (*int*) – The network's database identifier

**Return type** *Network*

**get_graph_by_id**(*network_id*)
Gets a network from the database by its identifier and converts it to a BEL graph

**Parameters** **network_id** (*int*) – The network's database identifier

**Return type** *BELGraph*

**get_networks_by_ids**(*network_ids*)
Gets a list of networks with the given identifiers. Note: order is not necessarily preserved.

**Parameters** **network_ids** (*iter[int]*) – The identifiers of networks in the database

**Return type** list[*Network*]

**get_graphs_by_ids**(*network_ids*)
Gets a list of networks with the given identifiers and converts to BEL graphs. Note: order is not necessarily preserved.

**Parameters** **network_ids** (*iter[int]*) – The identifiers of networks in the database

**Return type** list[*BELGraph*]

**get_graph_by_ids**(*network_ids*)
Gets a combine BEL Graph from a list of network identifiers

**Parameters** **network_ids** (*list[int]*) – A list of network identifiers

**Return type** *BELGraph*

**class** pybel.manager.**QueryManager**(*connection=None*, *echo=False*, *autoflush=None*, *autocommit=None*, *expire_on_commit=None*, *scopefunc=None*)
Groups queries over the edge store

**Parameters**

- **connection** (*str*) – An RFC-1738 database connection string. If `None`, tries to load from the environment variable `PYBEL_CONNECTION` then from the config file `~/.config/pybel/config.json` whose value for `PYBEL_CONNECTION` defaults to *pybel.constants.DEFAULT_CACHE_LOCATION*
- **echo** (*bool*) – Turn on echoing sql
- **autoflush** (*bool*) – Defaults to True if not specified in kwargs or configuration.
- **autocommit** (*bool*) – Defaults to False if not specified in kwargs or configuration.
- **expire_on_commit** (*bool*) – Defaults to False if not specified in kwargs or configuration.
- **scopefunc** – Scoped function to pass to `sqlalchemy.orm.scoped_session()`

From the Flask-SQLAlchemy documentation:

An extra key `'scopefunc'` can be set on the `options` dict to specify a custom scope function. If it's not provided, Flask's app context stack identity is used. This will ensure that sessions are created and removed with the request/response cycle, and should be fine in most cases.

**count_nodes**()
Counts the number of nodes in the cache

**Return type** int

**get_node_tuple_by_hash**(*node_hash*)
Looks up a node by the hash and returns the corresponding PyBEL node tuple

**Parameters** **node_hash** (*str*) – The hash of a PyBEL node tuple from `pybel.utils.hash_node()`

**Return type** Optional[tuple]

**get_node_by_tuple**(*node*)
Looks up a node by the PyBEL node tuple

**Parameters** **node** (*tuple*) – A PyBEL node tuple

**Return type** Optional[*Node*]

**query_nodes**(*bel=None*, *type=None*, *namespace=None*, *name=None*)
Builds and runs a query over all nodes in the PyBEL cache.

**Parameters**

- **bel** (*str*) – BEL term that describes the biological entity. e.g. `p(HGNC:APP)`
- **type** (*str*) – Type of the biological entity. e.g. Protein
- **namespace** (*str*) – Namespace keyword that is used in BEL. e.g. HGNC
- **name** (*str*) – Name of the biological entity. e.g. APP

**Return type** list[*Node*]

**count_edges**()
Counts the number of edges in the cache

**Return type** int

**get_edges_with_citation**(*citation*)
Gets the edges with the given citation

**Parameters** **citation** (`Citation`) –

**Return type** iter[*Edge*]

**get_edges_with_citations**(*citations*)
Gets the edges with the given citations

**Parameters** **citations** (*iter*[`Citation`]) –

**Return type** list[*Edge*]

**search_edges_with_evidence**(*evidence*)
Searches edges with the given evidence

**Parameters** **evidence** (*str*) – A string to search evidences. Can use wildcard percent symbol (%).

**Return type** list[*Edge*]

**search_edges_with_bel**(*bel*)
Searches edges with given BEL

**Parameters** **bel** (*str*) – A BEL string to use as a search

**Return type** list[*Edge*]

**get_edges_with_annotation**(*annotation*, *value*)

**Parameters**

- **annotation** (*str*) –
- **value** (*str*) –

**Return type** list[*Edge*]

**query_edges**(*bel=None*, *source_function=None*, *source=None*, *target_function=None*, *target=None*, *relation=None*)

Builds and runs a query over all edges in the PyBEL cache.

**Parameters**

- **bel** (*str*) – BEL statement that represents the desired edge.
- **source_function** (*str*) – Filter source nodes with the given BEL function
- **source** (*str or* Node) – BEL term of source node e.g. `p(HGNC:APP)` or Node object.
- **target_function** (*str*) – Filter target nodes with the given BEL function
- **target** (*str or* Node) – BEL term of target node e.g. `p(HGNC:APP)` or Node object.
- **relation** (*str*) – The relation that should be present between source and target node.

**Return type** list[*Edge*]

**query_citations**(*type=None*, *reference=None*, *name=None*, *author=None*, *date=None*, *evidence_text=None*)

Builds and runs a query over all citations in the PyBEL cache.

**Parameters**

- **type** (*str*) – Type of the citation. e.g. PubMed
- **reference** (*str*) – The identifier used for the citation. e.g. PubMed_ID
- **name** (*str*) – Title of the citation.
- **or list[str] author** (*str*) – The name or a list of names of authors participated in the citation.
- **date** (*str or datetime.date*) – Publishing date of the citation.
- **evidence_text** (*str*) –

**Return type** list[*Citation*]

**query_edges_by_pubmed_identifiers**(*pubmed_identifiers*)

Gets all edges annotated to the given documents

**Parameters** **pubmed_identifiers** (*list[str]*) – A list of PubMed document identifiers

**Return type** list[*Edge*]

**query_induction**(*nodes*)

Gets all edges between any of the given nodes

**Parameters** **nodes** (*list[*Node*]*) – A list of nodes (length > 2)

**Return type** list[*Edge*]

**query_neighbors**(*nodes*)

Gets all edges incident to any of the given nodes

**Parameters** **nodes** (*list[*Node*]*) – A list of nodes

**Return type** list[*Edge*]

## 2.10 Models

This module contains the SQLAlchemy database models that support the definition cache and graph cache.

**class** `pybel.manager.models.`**`Base`**`(`***kwargs*`)`

The most base type

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**class** `pybel.manager.models.`**`Namespace`**`(`***kwargs*`)`

Represents a BEL Namespace

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**`uploaded`**
The date of upload

**`keyword`**
Keyword that is used in a BEL file to identify a specific namespace

**`pattern`**
Contains regex pattern for value identification.

**`url`**
BELNS Resource location as URL

**`name`**
Name of the given namespace

**`domain`**
Domain for which this namespace is valid

**`species`**
Taxonomy identifiers for which this namespace is valid

**`description`**
Optional short description of the namespace

**`version`**
Version of the namespace

**`created`**
DateTime of the creation of the namespace definition file

**`query_url`**
URL that can be used to query the namespace (externally from PyBEL)

**`author`**
The author of the namespace

**`license`**
License information

**contact**
Contact information

**to_values**()
Returns this namespace as a dictionary of names to their encodings. Encodings are represented as a string, and lookup operations take constant time O(8).

**Return type** dict[str,str]

**to_tree_list**()
Returns an edge set of the tree represented by this namespace's hierarchy

**Return type** set[tuple[str,str]]

**to_json**(*include_id=False*)
Returns the most useful entries as a dictionary

**Parameters** **include_id** (*bool*) – If true, includes the model identifier

**Return type** dict[str,str]

**class** pybel.manager.models.**NamespaceEntry**(***kwargs*)
Represents a name within a BEL namespace

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**name**
Name that is defined in the corresponding namespace definition file

**identifier**
The database accession number

**encoding**
The biological entity types for which this name is valid

**to_json**(*include_id=False*)
Describes the namespaceEntry as dictionary of Namespace-Keyword and Name.

**Parameters** **include_id** (*bool*) – If true, includes the model identifier

**Return type** dict[str,str]

**class** pybel.manager.models.**NamespaceEntryEquivalence**(***kwargs*)
Represents the equivalance classes between entities

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**class** pybel.manager.models.**Annotation**(***kwargs*)
Represents a BEL Annotation

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**uploaded**
The date of upload

**url**
Source url of the given annotation definition file (.belanno)

**keyword**
Keyword that is used in a BEL file to identify a specific annotation

**type**
Annotation type

**description**
Optional short description of the given annotation

**version**
Version of the annotation

**created**
DateTime of the creation of the given annotation definition

**name**
Name of the annotation definition

**author**
Author information

**license**
License information

**contact**
Contact information

**get_entries**()
Gets a set of the names of all entries

**Return type** set[str]

**to_tree_list**()
Returns an edge set of the tree represented by this namespace's hierarchy

**Return type** set[tuple[str,str]]

**to_json**(*include_id=False*)
Returns this annotation as a JSON dictionary

**Parameters** **include_id** (*bool*) – If true, includes the model identifier

**Return type** dict[str,str]

**class** pybel.manager.models.**AnnotationEntry**(***kwargs*)
Represents a value within a BEL Annotation

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**name**
Name that is defined in the corresponding annotation definition file

**to_json**(*include_id=False*)
Describes the annotationEntry as dictionary of Annotation-Keyword and Annotation-Name.

**Parameters** **include_id** (*bool*) – If true, includes the model identifier

**Return type** dict[str,str]

**class** pybel.manager.models.**Network**(***kwargs*)

Represents a collection of edges, specified by a BEL Script

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**name**
Name of the given Network (from the BEL file)

**version**
Release version of the given Network (from the BEL file)

**authors**
Authors of the underlying BEL file

**contact**
Contact email from the underlying BEL file

**description**
Descriptive text from the underlying BEL file

**copyright**
Copyright information

**disclaimer**
Disclaimer information

**licenses**
License information

**blob**
A pickled version of this network

**to_json**(*include_id=False*)
Returns this network as JSON

**Parameters** **include_id** (*bool*) – If true, includes the model identifier

**Return type** dict[str,str]

**as_bel**()
Gets this network and loads it into a `BELGraph`

**Return type** *pybel.BELGraph*

**store_bel**(*graph*)
Inserts a bel graph

**Parameters** **graph** (`pybel.BELGraph`) – A BEL Graph

**class** pybel.manager.models.**Node**(***kwargs*)

Represents a BEL Term

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**`type`**
The type of the represented biological entity e.g. Protein or Gene

**`is_variant`**
Identifies weather or not the given node is a variant

**`has_fusion`**
Identifies weather or not the given node is a fusion

**`bel`**
Canonical BEL term that represents the given node

**`to_json`**(*include_id=False*, *include_hash=False*)
Serializes this node as a PyBEL node data dictionary

**Parameters**

- **`include_id`** (*`bool`*) – Include the database identifier?
- **`include_hash`** (*`bool`*) – Include the node hash?

**Return type** dict[str,str]

**`to_tuple`**()
Converts this node to a PyBEL tuple

**Return type** tuple

**class** `pybel.manager.models.`**`Modification`**(***kwargs*)
The modifications that are present in the network are stored in this table.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**`type`**
Type of the stored modification e.g. Fusion, gmod, pmod, etc

**`variantString`**
HGVS string if sequence modification

**`residue`**
Three letter amino acid code if PMOD

**`position`**
Position of PMOD or GMOD

**`to_json`**()
Recreates a is_variant dictionary for `BELGraph`

**Returns** Dictionary that describes a variant or a fusion.

**Return type** dict

**class** `pybel.manager.models.`**`Author`**(***kwargs*)
Contains all author names.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**class** `pybel.manager.models.`**`Citation`**`(`***kwargs*`)`

The information about the citations that are used to prove a specific relation are stored in this table.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**`type`**
Type of the stored publication e.g. PubMed

**`reference`**
Reference identifier of the publication e.g. PubMed_ID

**`name`**
Journal name

**`title`**
Title of the publication

**`volume`**
Volume of the journal

**`issue`**
Issue within the volume

**`pages`**
Pages of the publication

**`date`**
Publication date

**`first_id`**
First author

**`last_id`**
Last author

**`to_json`**`(`*include_id=False*`)`
Creates a citation dictionary that is used to recreate the edge data dictionary of a `BELGraph`.

**Parameters** **`include_id`** (*`bool`*) – If true, includes the model identifier

**Returns** Citation dictionary for the recreation of a `BELGraph`.

**Return type** dict[str,str]

**class** `pybel.manager.models.`**`Evidence`**`(`***kwargs*`)`

This table contains the evidence text that proves a specific relationship and refers the source that is cited.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**`text`**
Supporting text from a given publication

**to_json**(*include_id=False*)
Creates a dictionary that is used to recreate the edge data dictionary for a `BELGraph`.

**Parameters** **include_id** (*bool*) – If true, includes the model identifier

**Returns** Dictionary containing citation and evidence for a `BELGraph` edge.

**Return type** dict

**class** `pybel.manager.models.`**Edge**(***kwargs*)
Relationships are represented in this table. It shows the nodes that are in a relation to eachother and provides information about the context of the relation by refaring to the annotation, property and evidence tables.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**bel**
Valid BEL statement that represents the given edge

**sha512**
The hash of the source, target, and associated metadata

**get_annotations_json**()
Formats the annotations properly

**Return type** Optional[dict[str,dict[str,bool]]]

**get_data_json**()
Gets the PyBEL edge data dictionary this edge represents

**Return type** dict

**to_json**(*include_id=False*, *include_hash=False*)
Creates a dictionary of one BEL Edge that can be used to create an edge in a `BELGraph`.

**Parameters**

- **include_id** (*bool*) – Include the database identifier?
- **include_hash** (*bool*) – Include the node hash?

**Returns** Dictionary that contains information about an edge of a `BELGraph`. Including participants and edge data information.

**Return type** dict

**insert_into_graph**(*graph*)
Inserts this edge into a BEL Graph

**Parameters** **graph** (`pybel.BELGraph`) – A BEL graph

**class** `pybel.manager.models.`**Property**(***kwargs*)
The property table contains additional information that is used to describe the context of a relation.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

**is_subject**
Identifies which participant of the edge if affected by the given property

**modifier**
The modifier: one of activity, degradation, location, or translocation

**relative_key**
Relative key of effect e.g. to_tloc or from_tloc

**side**
Returns either *pybel.constants.SUBJECT* or *pybel.constants.OBJECT*

**Return type** str

**to_json**()
Creates a property dict that is used to recreate an edge dictionary for a `BELGraph`.

**Returns** Property dictionary of an edge that is participant (sub/obj) related.

**Return type** dict

## 2.11 Cookbook

An extensive set of examples can be found on the PyBEL Notebooks repository on GitHub. These notebooks contain basic usage and also make numerous references to the analytical package PyBEL Tools

### 2.11.1 Configuration

The default connection string can be set as an environment variable in your `~/.bashrc`. If you're using MySQL or MariaDB, it could look like this:

```
$ export PYBEL_CONNECTION="mysql+pymysql://user:password@server_name/database_name?
↪charset=utf8"
```

### 2.11.2 Command Line

**Note:** The command line wrapper might not work on Windows. Use `python3 -m pybel` if it has issues.

PyBEL automatically installs the command `pybel`. This command can be used to easily compile BEL documents and convert to other formats. See `pybel --help` for usage details. This command makes logs of all conversions and warnings to the directory `~/.pybel/`.

#### Prepare a Cytoscape Network

Load, compile, and export to Cytoscape format:

```
$ pybel convert --path ~/Desktop/example.bel --graphml ~/Desktop/example.graphml
```

In Cytoscape, open with `Import > Network > From File`.

## 2.12 Troubleshooting

Common problems and questions will be posted here.

### 2.12.1 Encoding Issues

Sometimes, Windows computers stick a weird unicode object `\u2013` at the beginning of files. This makes the function `pybel.parser.utils.sanitize_file_lines()` have a problem. The solution, when loading a BEL script via *pybel.from_path()* is to use the `encodings` keyword argument to specify the right encoding. The default is `utf-8` because this is the most common, but when this error happens, set it explicitly to `utf_8_sig`. More specific documentation is available in the Inputs and Outputs page.

#### Scenario

```
>>> import pybel
>>> graph = pybel.from_path('~/Desktop/small_corpus.bel')

UnicodeDecodeError Traceback (most recent call last) <ipython-input-11-99f2a76596b1>␣
↪in <module>()
7 ad = pybel.from_pickle(path_2_AD_pickle)
8 else:
----> 9 ad = pybel.from_path(path_2_AD_bel)
10 pybel.to_pickle(ad, path_2_AD_pickle)

C:\Users\s8310253\AppData\Local\Continuum\Anaconda3420\lib\site-packages\pybel\graph.
↪py in from_path(path, **kwargs)
61 log.info('Loading from path: %s', path)
62 with open(os.path.expanduser(path)) as f:
---> 63 return BELGraph(lines=f, **kwargs)
64
65

C:\Users\s8310253\AppData\Local\Continuum\Anaconda3420\lib\site-packages\pybel\graph.
↪py in __init__(self, lines, context, lenient, definition_cache_manager, log_stream,␣
↪*attrs, **kwargs)
102
103 if lines is not None:
--> 104 self.parse_lines(lines, context, lenient,
--> definition_cache_manager, log_stream)
105
106 def parse_lines(self, lines, context=None, lenient=False, definition_cache_
↪manager=None, log_stream=None):

C:\Users\s8310253\AppData\Local\Continuum\Anaconda3420\lib\site-packages\pybel\graph.
↪py in parse_lines(self, lines, context, lenient, definition_cache_manager, log_
↪stream)
125 self.context = context
126
--> 127 docs, defs, states =
--> split_file_to_annotations_and_definitions(lines)
128
129 if isinstance(definition_cache_manager, DefinitionCacheManager):

C:\Users\s8310253\AppData\Local\Continuum\Anaconda3420\lib\site-
↪packages\pybel\parser\utils.py in split_file_to_annotations_and_definitions(file)
49 def split_file_to_annotations_and_definitions(file):
50 """Enumerates a line iterable and splits into 3 parts"""
---> 51 content = list(sanitize_file_lines(file))
52
53 end_document_section = 1 + max(j for j, (i, l) in enumerate(content) if l.
↪startswith('SET DOCUMENT'))
```

```
C:\Users\s8310253\AppData\Local\Continuum\Anaconda3420\lib\site-
↪packages\pybel\parser\utils.py in sanitize_file_lines(f)
16 it = iter(it)
17
---> 18 for line_number, line in it:
19 if line.endswith('\\'):
20 log.log(4, 'Multiline quote starting on line: %d', line_number)

C:\Users\s8310253\AppData\Local\Continuum\Anaconda3420\lib\site-
↪packages\pybel\parser\utils.py in <genexpr>(.0)
12 def sanitize_file_lines(f):
13 """Enumerates a line iterator and returns the pairs of (line number, line) that␣
↪are cleaned"""
---> 14 it = (line.strip() for line in f)
15 it = filter(lambda i_l: i_l[1] and not i_l[1].startswith('#'), enumerate(it,␣
↪start=1))
16 it = iter(it)

C:\Users\s8310253\AppData\Local\Continuum\Anaconda3420\lib\encodings\cp1252.py in␣
↪decode(self, input, final)
21 class IncrementalDecoder(codecs.IncrementalDecoder):
22 def decode(self, input, final=False):
---> 23 return
---> codecs.charmap_decode(input,self.errors,decoding_table)[0]
24
25 class StreamWriter(Codec,codecs.StreamWriter):

UnicodeDecodeError: 'charmap' codec can't decode byte 0x9d in position 4668:␣
↪character maps to <undefined>
```

#### Solution

```
>>> import pybel
>>> graph = pybel.from_path('~/Desktop/small_corpus.bel', encoding='utf_8_sig')
>>> # Success!
```

## 2.13 Constants

### 2.13.1 PyBEL Constants

This module maintains the strings used throughout the PyBEL codebase to promote consistency.

### 2.13.2 Configuration Loading

By default, PyBEL loads its configuration from `~/.config/pybel/config.json`. This json is stored in the object *`pybel.constants.config`*.

`pybel.constants.`**`PYBEL_MINIMUM_IMPORT_VERSION = (0, 11, 0)`**
The last PyBEL version where the graph data definition changed

pybel.constants.**GOCC_LATEST = 'https://arty.scai.fraunhofer.de/artifactory/bel/namespace/go**
GOCC is the only namespace that needs to be stored because translocations use some of its values by default

pybel.constants.**PYBEL_CONNECTION = 'PYBEL_CONNECTION'**
The environment variable that contains the default SQL connection information for the PyBEL cache

pybel.constants.**PYBEL_DIR = '/home/docs/.pybel'**
The default directory where PyBEL files, including logs and the default cache, are stored. Created if not exists.

pybel.constants.**DEFAULT_CACHE_LOCATION = '/home/docs/.pybel/pybel_0.11.0_cache.db'**
The default cache location is `~/.pybel/data/pybel_cache.db`

pybel.constants.**DEFAULT_CACHE_CONNECTION = 'sqlite:////home/docs/.pybel/pybel_0.11.0_cache**
The default cache connection string uses sqlite.

pybel.constants.**config = {'PYBEL_CONNECTION': 'sqlite:////home/docs/.pybel/pybel_0.11.0_cac**
The global configuration for PyBEL is stored here. By default, it loads from `~/.config/pybel/config.json`

pybel.constants.**get_cache_connection**(*connection=None*)
Returns the default cache connection string. If a connection string is explicitly given, passes it through

**Parameters** **connection** (*str*) – RFC connection string

**Return type** str

pybel.constants.**BEL_DEFAULT_NAMESPACE = 'bel'**
The default namespace given to entities in the BEL language

pybel.constants.**CITATION_TYPES = {'DOI', 'Journal', 'URL', 'Other', 'Online Resource', 'Pub**
The valid citation types .. seealso:: https://wiki.openbel.org/display/BELNA/Citation

pybel.constants.**NAMESPACE_DOMAIN_TYPES = {'Chemical', 'BiologicalProcess', 'Other', 'Gene a**
The valid namespace types .. seealso:: https://wiki.openbel.org/display/BELNA/Custom+Namespaces

pybel.constants.**CITATION_TYPE = 'type'**
Represents the key for the citation type in a citation dictionary

pybel.constants.**CITATION_NAME = 'name'**
Represents the key for the citation name in a citation dictionary

pybel.constants.**CITATION_REFERENCE = 'reference'**
Represents the key for the citation reference in a citation dictionary

pybel.constants.**CITATION_DATE = 'date'**
Represents the key for the citation date in a citation dictionary

pybel.constants.**CITATION_AUTHORS = 'authors'**
Represents the key for the citation authors in a citation dictionary

pybel.constants.**CITATION_COMMENTS = 'comments'**
Represents the key for the citation comment in a citation dictionary

pybel.constants.**CITATION_TITLE = 'title'**
Represents the key for the optional PyBEL citation title entry in a citation dictionary

pybel.constants.**CITATION_VOLUME = 'volume'**
Represents the key for the optional PyBEL citation volume entry in a citation dictionary

pybel.constants.**CITATION_ISSUE = 'issue'**
Represents the key for the optional PyBEL citation issue entry in a citation dictionary

pybel.constants.**CITATION_PAGES = 'pages'**
Represents the key for the optional PyBEL citation pages entry in a citation dictionary

pybel.constants.**CITATION_FIRST_AUTHOR = 'first'**
Represents the key for the optional PyBEL citation first author entry in a citation dictionary

pybel.constants.**CITATION_LAST_AUTHOR = 'last'**
Represents the key for the optional PyBEL citation last author entry in a citation dictionary

pybel.constants.**CITATION_ENTRIES = ('type', 'name', 'reference', 'date', 'authors', 'commen**
Represents the ordering of the citation entries in a control statement (SET Citation = . . . )

pybel.constants.**FUNCTION = 'function'**
The node data key specifying the node's function (e.g. *GENE*, *MIRNA*, *BIOPROCESS*, etc.)

pybel.constants.**NAMESPACE = 'namespace'**
The key specifying an identifier dictionary's namespace. Used for nodes, activities, and transformations.

pybel.constants.**NAME = 'name'**
The key specifying an identifier dictionary's name. Used for nodes, activities, and transformations.

pybel.constants.**IDENTIFIER = 'identifier'**
The key specifying an identifier dictionary

pybel.constants.**LABEL = 'label'**
The key specifying an optional label for the node

pybel.constants.**DESCRIPTION = 'description'**
The key specifying an optional description for the node

pybel.constants.**MEMBERS = 'members'**
They key representing the nodes that are a member of a composite or complex

pybel.constants.**REACTANTS = 'reactants'**
The key representing the nodes appearing in the reactant side of a biochemical reaction

pybel.constants.**PRODUCTS = 'products'**
The key representing the nodes appearing in the product side of a biochemical reaction

pybel.constants.**FUSION = 'fusion'**
The node data key specifying a fusion dictionary, containing *PARTNER_3P*, *PARTNER_5P*,

pybel.constants.**PARTNER_3P = 'partner_3p'**
The key specifying the identifier dictionary of the fusion's 3-Prime partner

pybel.constants.**PARTNER_5P = 'partner_5p'**
The key specifying the identifier dictionary of the fusion's 5-Prime partner

pybel.constants.**RANGE_3P = 'range_3p'**
The key specifying the range dictionary of the fusion's 3-Prime partner

pybel.constants.**RANGE_5P = 'range_5p'**
The key specifying the range dictionary of the fusion's 5-Prime partner

pybel.constants.**VARIANTS = 'variants'**
The key specifying the node has a list of associated variants

pybel.constants.**KIND = 'kind'**
The key representing what kind of variation is being represented

pybel.constants.**HGVS = 'hgvs'**
The value for *KIND* for an HGVS variant

pybel.constants.**PMOD = 'pmod'**
The value for *KIND* for a protein modification

`pybel.constants.`**`GMOD = 'gmod'`**
The value for *KIND* for a gene modification

`pybel.constants.`**`FRAGMENT = 'frag'`**
The value for *KIND* for a fragment

`pybel.constants.`**`PYBEL_VARIANT_KINDS = {'hgvs', 'frag', 'gmod', 'pmod'}`**
The allowed values for *KIND*

`pybel.constants.`**`PYBEL_NODE_DATA_KEYS = {'reactants', 'variants', 'function', 'fusion', 'nar`**
The group of all BEL-provided keys for node data dictionaries, used for hashing.

`pybel.constants.`**`DIRTY = 'dirty'`**
Used as a namespace when none is given when lenient parsing mode is turned on. Not recommended!

`pybel.constants.`**`ABUNDANCE = 'Abundance'`**
Represents the BEL abundance, abundance()

`pybel.constants.`**`GENE = 'Gene'`**
Represents the BEL abundance, geneAbundance() .. seealso:: http://openbel.org/language/version_2.0/bel_specification_version_2.0.html#Xabundancea

`pybel.constants.`**`RNA = 'RNA'`**
Represents the BEL abundance, rnaAbundance()

`pybel.constants.`**`MIRNA = 'miRNA'`**
Represents the BEL abundance, microRNAAbundance()

`pybel.constants.`**`PROTEIN = 'Protein'`**
Represents the BEL abundance, proteinAbundance()

`pybel.constants.`**`BIOPROCESS = 'BiologicalProcess'`**
Represents the BEL function, biologicalProcess()

`pybel.constants.`**`PATHOLOGY = 'Pathology'`**
Represents the BEL function, pathology()

`pybel.constants.`**`COMPOSITE = 'Composite'`**
Represents the BEL abundance, compositeAbundance()

`pybel.constants.`**`COMPLEX = 'Complex'`**
Represents the BEL abundance, complexAbundance()

`pybel.constants.`**`REACTION = 'Reaction'`**
Represents the BEL transformation, reaction()

`pybel.constants.`**`PYBEL_NODE_FUNCTIONS = {'Protein', 'RNA', 'Reaction', 'BiologicalProcess',`**
A set of all of the valid PyBEL node functions

`pybel.constants.`**`rev_abundance_labels = {'Composite': 'composite', 'Protein': 'p', 'Comple`**
The mapping from PyBEL node functions to BEL strings

`pybel.constants.`**`RELATION = 'relation'`**
The key for an internal edge data dictionary for the relation string

`pybel.constants.`**`CITATION = 'citation'`**
The key for an internal edge data dictionary for the citation dictionary

`pybel.constants.`**`EVIDENCE = 'evidence'`**
The key for an internal edge data dictionary for the evidence string

`pybel.constants.`**`ANNOTATIONS = 'annotations'`**
The key for an internal edge data dictionary for the annotations dictionary

`pybel.constants.`**`SUBJECT = 'subject'`**
The key for an internal edge data dictionary for the subject modifier dictionary

`pybel.constants.`**`OBJECT = 'object'`**
The key for an internal edge data dictionary for the object modifier dictionary

`pybel.constants.`**`LINE = 'line'`**
The key or an internal edge data dictionary for the line number

`pybel.constants.`**`HASH = 'hash'`**
The key representing the hash of the other

`pybel.constants.`**`PYBEL_EDGE_DATA_KEYS = {'relation', 'evidence', 'citation', 'annotations',`**
The group of all BEL-provided keys for edge data dictionaries, used for hashing.

`pybel.constants.`**`PYBEL_EDGE_METADATA_KEYS = {'hash', 'line'}`**
The group of all PyBEL-specific keys for edge data dictionaries, not used for hashing.

`pybel.constants.`**`PYBEL_EDGE_ALL_KEYS = {'evidence', 'object', 'citation', 'annotations', 're`**
The group of all PyBEL annotated keys for edge data dictionaries

`pybel.constants.`**`HAS_REACTANT = 'hasReactant'`**
A BEL relationship

`pybel.constants.`**`HAS_PRODUCT = 'hasProduct'`**
A BEL relationship

`pybel.constants.`**`HAS_COMPONENT = 'hasComponent'`**
A BEL relationship

`pybel.constants.`**`HAS_VARIANT = 'hasVariant'`**
A BEL relationship

`pybel.constants.`**`HAS_MEMBER = 'hasMember'`**
A BEL relationship

`pybel.constants.`**`TRANSCRIBED_TO = 'transcribedTo'`**
A BEL relationship *`GENE`* to *`RNA`* is called transcription

`pybel.constants.`**`TRANSLATED_TO = 'translatedTo'`**
A BEL relationship *`RNA`* to *`PROTEIN`* is called translation

`pybel.constants.`**`INCREASES = 'increases'`**
A BEL relationship

`pybel.constants.`**`DIRECTLY_INCREASES = 'directlyIncreases'`**
A BEL relationship

`pybel.constants.`**`DECREASES = 'decreases'`**
A BEL relationship

`pybel.constants.`**`DIRECTLY_DECREASES = 'directlyDecreases'`**
A BEL relationship

`pybel.constants.`**`CAUSES_NO_CHANGE = 'causesNoChange'`**
A BEL relationship

`pybel.constants.`**`REGULATES = 'regulates'`**
A BEL relationship

`pybel.constants.`**`NEGATIVE_CORRELATION = 'negativeCorrelation'`**
A BEL relationship

`pybel.constants.`**`POSITIVE_CORRELATION = 'positiveCorrelation'`**
A BEL relationship

`pybel.constants.`**`ASSOCIATION = 'association'`**
A BEL relationship

`pybel.constants.`**`ORTHOLOGOUS = 'orthologous'`**
A BEL relationship

`pybel.constants.`**`ANALOGOUS_TO = 'analogousTo'`**
A BEL relationship

`pybel.constants.`**`IS_A = 'isA'`**
A BEL relationship

`pybel.constants.`**`RATE_LIMITING_STEP_OF = 'rateLimitingStepOf'`**
A BEL relationship

`pybel.constants.`**`SUBPROCESS_OF = 'subProcessOf'`**
A BEL relationship

`pybel.constants.`**`BIOMARKER_FOR = 'biomarkerFor'`**
A BEL relationship

`pybel.constants.`**`PROGONSTIC_BIOMARKER_FOR = 'prognosticBiomarkerFor'`**
A BEL relationship

`pybel.constants.`**`EQUIVALENT_TO = 'equivalentTo'`**
A BEL relationship, added by PyBEL

`pybel.constants.`**`PART_OF = 'partOf'`**
A BEL relationship, added by PyBEL

`pybel.constants.`**`CAUSAL_INCREASE_RELATIONS = {'increases', 'directlyIncreases'}`**
A set of all causal relationships that have an increasing effect

`pybel.constants.`**`CAUSAL_DECREASE_RELATIONS = {'decreases', 'directlyDecreases'}`**
A set of all causal relationships that have a decreasing effect

`pybel.constants.`**`DIRECT_CAUSAL_RELATIONS = {'directlyDecreases', 'directlyIncreases'}`**
A set of direct causal relations

`pybel.constants.`**`INDIRECT_CAUSAL_RELATIONS = {'decreases', 'increases'}`**
A set of direct causal relations

`pybel.constants.`**`CAUSAL_RELATIONS = {'decreases', 'directlyDecreases', 'increases', 'directl`**
A set of all causal relationships

`pybel.constants.`**`TWO_WAY_RELATIONS = {'analogousTo', 'association', 'orthologous', 'equivale`**
A set of all relationships that are inherently directionless, and are therefore added to the graph twice

`pybel.constants.`**`CORRELATIVE_RELATIONS = {'negativeCorrelation', 'positiveCorrelation'}`**
A set of all correlative relationships

`pybel.constants.`**`POLAR_RELATIONS = {'decreases', 'positiveCorrelation', 'negativeCorrelatio`**
A set of polar relations

`pybel.constants.`**`unqualified_edges = ['hasReactant', 'hasProduct', 'hasComponent', 'hasVari`**
A list of relationship types that don't require annotations or evidence This must be maintained as a list, since the *`unqualified_edge_code`* is calculated based on the order and needs to be consistent

`pybel.constants.`**`unqualified_edge_code = {'hasComponent': -3, 'orthologous': -11, 'transc`**
Unqualified edges are given negative keys since the standard NetworkX edge key factory starts at 0 and counts up

pybel.constants.**GRAPH_METADATA = 'document_metadata'**
: The key for the document metadata dictionary. Can be accessed by `graph.graph[GRAPH_METADATA]`, or by using the property built in to the *pybel.BELGraph*, *pybel.BELGraph.document()*

pybel.constants.**METADATA_NAME = 'name'**
: The key for the document name. Can be accessed by `graph.document[METADATA_NAME]` or by using the property built into the *pybel.BELGraph* class, *pybel.BELGraph.name()*

pybel.constants.**METADATA_VERSION = 'version'**
: The key for the document version. Can be accessed by `graph.document[METADATA_VERSION]`

pybel.constants.**METADATA_DESCRIPTION = 'description'**
: The key for the document description. Can be accessed by `graph.document[METADATA_DESCRIPTION]`

pybel.constants.**METADATA_AUTHORS = 'authors'**
: The key for the document authors. Can be accessed by `graph.document[METADATA_NAME]`

pybel.constants.**METADATA_CONTACT = 'contact'**
: The key for the document contact email. Can be accessed by `graph.document[METADATA_CONTACT]`

pybel.constants.**METADATA_LICENSES = 'licenses'**
: The key for the document licenses. Can be accessed by `graph.document[METADATA_LICENSES]`

pybel.constants.**METADATA_COPYRIGHT = 'copyright'**
: The key for the document copyright information. Can be accessed by `graph.document[METADATA_COPYRIGHT]`

pybel.constants.**METADATA_DISCLAIMER = 'disclaimer'**
: The key for the document disclaimer. Can be accessed by `graph.document[METADATA_DISCLAIMER]`

pybel.constants.**METADATA_PROJECT = 'project'**
: The key for the document project. Can be accessed by `graph.document[METADATA_PROJECT]`

pybel.constants.**DOCUMENT_KEYS = {'Copyright': 'copyright', 'Name': 'name', 'Licenses':**
: Provides a mapping from BEL language keywords to internal PyBEL strings

pybel.constants.**METADATA_INSERT_KEYS = {'copyright', 'description', 'contact', 'authors',**
: The keys to use when inserting a graph to the cache

pybel.constants.**INVERSE_DOCUMENT_KEYS = {'authors': 'Authors', 'project': 'Project', 'nam**
: Provides a mapping from internal PyBEL strings to BEL language keywords. Is the inverse of *DOCUMENT_KEYS*

pybel.constants.**REQUIRED_METADATA = {'contact', 'authors', 'name', 'version', 'description**
: A set representing the required metadata during BEL document parsing

pybel.constants.**FRAGMENT_START = 'start'**
: The key for the starting position of a fragment range

pybel.constants.**FRAGMENT_STOP = 'stop'**
: The key for the stopping position of a fragment range

pybel.constants.**FRAGMENT_MISSING = 'missing'**
: The key signifying that there is neither a start nor stop position defined

pybel.constants.**FRAGMENT_DESCRIPTION = 'description'**
: The key for any additional descriptive data about a fragment

pybel.constants.**GMOD_ORDER = ['kind', 'identifier']**
: The order for serializing gene modification data

`pybel.constants.GSUB_REFERENCE = 'reference'`
The key for the reference nucleotide in a gene substitution. Only used during parsing since this is converted to HGVS.

`pybel.constants.GSUB_POSITION = 'position'`
The key for the position of a gene substitution. Only used during parsing since this is converted to HGVS

`pybel.constants.GSUB_VARIANT = 'variant'`
The key for the effect of a gene substitution. Only used during parsing since this is converted to HGVS

`pybel.constants.PMOD_CODE = 'code'`
The key for the protein modification code.

`pybel.constants.PMOD_POSITION = 'pos'`
The key for the protein modification position.

`pybel.constants.PMOD_ORDER = ['kind', 'identifier', 'code', 'pos']`
The order for serializing information about a protein modification

`pybel.constants.PSUB_REFERENCE = 'reference'`
The key for the reference amino acid in a protein substitution. Only used during parsing since this is concerted to HGVS

`pybel.constants.PSUB_POSITION = 'position'`
The key for the position of a protein substitution. Only used during parsing since this is converted to HGVS.

`pybel.constants.PSUB_VARIANT = 'variant'`
The key for the variant of a protein substitution.Only used during parsing since this is converted to HGVS.

`pybel.constants.TRUNCATION_POSITION = 'position'`
The key for the position at which a protein is truncated

`pybel.constants.belns_encodings = {'C': {'Complex'}, 'G': {'Gene'}, 'M': {'miRNA'}, 'R': {`
The mapping from BEL namespace codes to PyBEL internal abundance constants ..seealso:: https://wiki.openbel.org/display/BELNA/Assignment+of+Encoding+%28Allowed+Functions%29+for+BEL+Namespaces

`pybel.constants.DEFAULT_SERVICE_URL = 'https://pybel.scai.fraunhofer.de'`
The default location of PyBEL Web

## 2.13.3 BEL Language

This module contains mappings between PyBEL's internal constants and BEL language keywords

`pybel.language.activity_labels = {'catalyticActivity': 'cat', 'kin': 'kin', 'gtpaseActiva`
A dictionary of activity labels used in the ma() function in activity(p(X), ma(Y))

`pybel.language.activity_mapping = {'cat': {'namespace': 'GO', 'name': 'catalytic activit`
Maps the default BEL molecular activities to Gene Ontology Molecular Functions

`pybel.language.abundance_labels = {'geneAbundance': 'Gene', 'p': 'Protein', 'rnaAbundance`
Provides a mapping from BEL terms to PyBEL internal constants

`pybel.language.abundance_sbo_mapping = {'RNA': {'namespace': 'SBO', 'name': 'messenger RN`
Maps the BEL abundance types to the Systems Biology Ontology

`pybel.language.pmod_namespace = {'sulfur addition': 'Sulf', 'N-linked glycosylation': 'NG`
A dictionary of default protein modifications to their preferred value

`pybel.language.pmod_mappings = {'Me3': {'xrefs': [{'namespace': 'MOD', 'name': 'trimeth`
Use Gene Ontology children of GO_0006464: "cellular protein modification process"

pybel.language.**pmod_legacy_labels = {'S': 'Sumo', 'F': 'Farn', 'M': 'Me', 'R': 'ADPRib', 'I**
A dictionary of legacy (BEL 1.0) default namespace protein modifications to their BEL 2.0 preferred value

pybel.language.**gmod_namespace = {'M': 'Me', 'Me': 'Me', 'methylation': 'Me'}**
A dictionary of default gene modifications. This is a PyBEL variant to the BEL specification.

pybel.language.**gmod_mappings = {'ADPRib': {'xrefs': [{'namespace': 'GO', 'name': 'DNA A**
Use Gene Ontology children of GO_0006304: "DNA modification"

## 2.14 Parsers

This page is for users who want to squeeze the most bizarre possibilities out of PyBEL. Most users will not need this reference.

PyBEL makes extensive use of the PyParsing module. The code is organized to different modules to reflect the different faces ot the BEL language. These parsers support BEL 2.0 and have some backwards compatibility for rewriting BEL v1.0 statements as BEL v2.0. The biologist and bioinformatician using this software will likely never need to read this page, but a developer seeking to extend the language will be interested to see the inner workings of these parsers.

See: https://github.com/OpenBEL/language/blob/master/version_2.0/MIGRATE_BEL1_BEL2.md

### 2.14.1 Metadata Parser

**class** pybel.parser.parse_metadata.**MetadataParser**(*manager*, *namespace_dict=None*, *annotation_dict=None*, *namespace_regex=None*, *annotation_regex=None*, *default_namespace=None*, *allow_redefinition=False*)
A parser for the document and definitions section of a BEL document.

**See also:**

BEL 1.0 Specification for the DEFINE keyword

**Parameters**

- **manager** (pybel.manager.Manager) – A cache manager
- **namespace_dict** (*dict[str, dict[str, str]]*) – A dictionary of pre-loaded, enumerated namespaces from {namespace keyword: {name: encoding}}
- **annotation_dict** (*dict[str, set[str]]*) – A dictionary of pre-loaded, enumerated annotations from {annotation keyword: set of valid values}
- **namespace_regex** (*dict[str, str]*) – A dictionary of pre-loaded, regular expression namespaces from {namespace keyword: regex string}
- **annotation_regex** (*dict[str, str]*) – A dictionary of pre-loaded, regular expression annotations from {annotation keyword: regex string}
- **default_namespace** (*set[str]*) – A set of strings that can be used without a namespace

**manager = None**
This metadata parser's internal definition cache manager

**namespace_dict = None**
A dictionary of cached {namespace keyword: {name: encoding}}

**annotation_dict = None**
A dictionary of cached {annotation keyword: set of values}

**namespace_regex = None**
A dictionary of {namespace keyword: regular expression string}

**default_namespace = None**
A set of names that can be used without a namespace

**annotation_regex = None**
A dictionary of {annotation keyword: regular expression string}

**uncachable_namespaces = None**
A set of namespaces's URLs that can't be cached

**document_metadata = None**
A dictionary containing the document metadata

**namespace_url_dict = None**
A dictionary from {namespace keyword: BEL namespace URL}

**namespace_owl_dict = None**
A dictionary from {namespace keyword: OWL namespace URL}

**annotation_url_dict = None**
A dictionary from {annotation keyword: BEL annotation URL}

**annotation_owl_dict = None**
A dictionary from {annotation keyword: OWL annotation URL}

**annotation_lists = None**
A set of annotation keywords that are defined ad-hoc in the BEL script

**handle_document** (*line*, *position*, *tokens*)
Handles statements like `SET DOCUMENT X = "Y"`

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**raise_for_redefined_namespace** (*line*, *position*, *namespace*)
Raises an exception if a namespace is already defined

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **namespace** (*str*) – The namespace being parsed

**Raises** RedefinedNamespaceError

**handle_namespace_url** (*line*, *position*, *tokens*)
Handles statements like `DEFINE NAMESPACE X AS URL "Y"`

**Parameters**

- **line** (*str*) – The line being parsed

- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** RedefinedNamespaceError

**Raises** pybel.resources.exc.ResourceError

**handle_namespace_owl**(*line*, *position*, *tokens*)
Handles statements like `DEFINE NAMESPACE X AS OWL "Y"`

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** RedefinedNamespaceError

**handle_namespace_pattern**(*line*, *position*, *tokens*)
Handles statements like `DEFINE NAMESPACE X AS PATTERN "Y"`

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** RedefinedNamespaceError

**raise_for_redefined_annotation**(*line*, *position*, *annotation*)
Raises an exception if the given annotation is already defined

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **annotation** (*str*) – The annotation being parsed

**Raises** RedefinedAnnotationError

**handle_annotation_owl**(*line*, *position*, *tokens*)
Handles statements like `DEFINE ANNOTATION X AS OWL "Y"`

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** RedefinedAnnotationError

**handle_annotations_url**(*line*, *position*, *tokens*)
Handles statements like `DEFINE ANNOTATION X AS URL "Y"`

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** RedefinedAnnotationError

**`handle_annotation_list`**(*line*, *position*, *tokens*)
Handles statements like `DEFINE ANNOTATION X AS LIST {"Y","Z", ...}`

**Parameters**

- **`line`** (*`str`*) – The line being parsed
- **`position`** (*`int`*) – The position in the line being parsed
- **`tokens`** (*`pyparsing.ParseResult`*) – The tokens from PyParsing

**Raises** RedefinedAnnotationError

**`handle_annotation_pattern`**(*line*, *position*, *tokens*)
Handles statements like `DEFINE ANNOTATION X AS PATTERN "Y"`

**Parameters**

- **`line`** (*`str`*) – The line being parsed
- **`position`** (*`int`*) – The position in the line being parsed
- **`tokens`** (*`pyparsing.ParseResult`*) – The tokens from PyParsing

**Raises** RedefinedAnnotationError

**`has_enumerated_annotation`**(*annotation*)
Checks if this annotation is defined by an enumeration

**Parameters** **`annotation`** (*`str`*) – The keyword of a annotation

**Return type** bool

**`has_regex_annotation`**(*annotation*)
Checks if this annotation is defined by a regular expression

**Parameters** **`annotation`** (*`str`*) – The keyword of a annotation

**Return type** bool

**`has_annotation`**(*annotation*)
Checks if this annotation is defined

**Parameters** **`annotation`** (*`str`*) – The keyword of a annotation

**Return type** bool

**`has_enumerated_namespace`**(*namespace*)
Checks if this namespace is defined by an enumeration

**Parameters** **`namespace`** (*`str`*) – The keyword of a namespace

**Return type** bool

**`has_regex_namespace`**(*namespace*)
Checks if this namespace is defined by a regular expression

**Parameters** **`namespace`** (*`str`*) – The keyword of a namespace

**Return type** bool

**`has_namespace`**(*namespace*)
Checks if this namespace is defined

**Parameters** **`namespace`** (*`str`*) – The keyword of a namespace

**Return type** bool

**raise_for_version**(*line*, *position*, *version*)

Checks that a version string is valid for BEL documents, meaning it's either in the YYYYMMDD or semantic version format

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **version** (*str*) – A version string

**Raises** VersionFormatWarning

### 2.14.2 Control Parser

**class** pybel.parser.parse_control.**ControlParser**(*annotation_dict=None*, *annotation_regex=None*, *citation_clearing=True*)

A parser for BEL control statements

**See also:**

BEL 1.0 specification on control records

**Parameters**

- **annotation_dict** (*dict[str, set[str]]*) – A dictionary of {annotation: set of valid values} for parsing
- **annotation_regex** (*dict[str, str]*) – A dictionary of {annotation: regular expression string}
- **citation_clearing** (*bool*) – Should `SET Citation` statements clear evidence and all annotations?

**annotation_dict**

A dictionary of annotaions to their set of values

**Return type** dict[str,set[str]]

**annotation_regex**

A dictioary of annotations defined by regular expressions {annotation keyword: string regular expression}

**Returns** dict[str,str]

**annotation_regex_compiled**

A dictionary of annotations defined by regular expressions {annotation keyword: compiled regular expression}

**Return type** dict[str,re]

**raise_for_undefined_annotation**(*line*, *position*, *annotation*)

Raises is an annotation is not defined

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **annotation** (*str*) – The annotation to check

**Raises** UndefinedAnnotationWarning

**raise_for_invalid_annotation_value**(*line*, *position*, *key*, *value*)
Raises is an annotation is not defined

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **key** (*str*) – The annotation to check
- **value** (*str*) – The entry in the annotation to check

**Raises** IllegalAnnotationValueWarning or MissingAnnotationRegexWarning

**raise_for_missing_citation**(*line*, *position*)
Raises if there is no citation present in the parser

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed

**Raises** MissingCitationException

**handle_annotation_key**(*line*, *position*, *tokens*)
Called on all annotation keys before parsing to validate that it's either enumerated or as a regex

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raise** MissingCitationException or UndefinedAnnotationWarning

**handle_unset_statement_group**(*line*, *position*, *tokens*)
Unsets the statement group, or raises an exception if it is not set.

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** MissingAnnotationKeyWarning

**handle_unset_citation**(*line*, *position*, *tokens*)
Unsets the citation, or raises an exception if it is not set

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** MissingAnnotationKeyWarning

**handle_unset_evidence**(*line*, *position*, *tokens*)
Unsets the evidence, or throws an exception if it is not already set. The value for `tokens[EVIDENCE]` corresponds to which alternate of SupportingText or Evidence was used in the BEL script.

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** MissingAnnotationKeyWarning

**validate_unset_command**(*line*, *position*, *key*)

Raises an exception when trying to `UNSET X` if `X` is not already set.

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **key** (*str*) – The annotation to check

**Raises** MissingAnnotationKeyWarning

**handle_unset_command**(*line*, *position*, *tokens*)

Handles `UNSET X` or raises an exception if it is not already set.

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** MissingAnnotationKeyWarning

**handle_unset_list**(*line*, *position*, *tokens*)

Handles `UNSET {A, B, ...}` or raises an exception of any of them are not present. Consider that all unsets are in peril if just one of them is wrong!

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** MissingAnnotationKeyWarning

**handle_unset_all**(*line*, *position*, *tokens*)

Handles `UNSET_ALL`

**get_annotations**()

Gets the current annotations

**Returns** The currently stored BEL annotations

**Return type** dict

**clear_citation**()

Clears the citation. Additionally, if citation clearing is enabled, clears the evidence and annotations.

**clear**()

Clears the statement_group, citation, evidence, and annotations

### 2.14.3 Identifier Parser

**class** `pybel.parser.parse_identifier.`**`IdentifierParser`**(*namespace_dict=None*, *namespace_regex=None*, *default_namespace=None*, *allow_naked_names=False*)

A parser for identifiers in the form of namespace:name. Can be made more lenient when given a default namespace or enabling the use of naked names

**Parameters**

- **`namespace_dict`** (*`Optional[dict[str, dict[str, str]]]`*) – A dictionary of {namespace: {name: encoding}}
- **`namespace_regex`** (*`Optional[dict[str, str]]`*) – A dictionary of {namespace: regular expression string} to compile
- **`default_namespace`** (*`Optional[set[str]]`*) – A set of strings that can be used without a namespace
- **`allow_naked_names`** (*`bool`*) – If true, turn off naked namespace failures

**`namespace_dict`**

A dictionary of {namespace: {name: encodings}}

**Return type** dict[str,dict[str,str]]

**`namespace_regex`**

A dictionary of {namespace keyword: regular expression string}

**Return type** dict[str,str]

**`namespace_regex_compiled`**

A dictionary of {namespace keyword: compiled regular expression}

**Return type** dict[str,re]

**`has_enumerated_namespace`**(*namespace*)

Checks that the namespace has been defined by an enumeration

**`has_regex_namespace`**(*namespace*)

Checks that the namespace has been defined by a regular expression

**`has_namespace`**(*namespace*)

Checks that the namespace has either been defined by an enumeration or a regular expression

**`has_enumerated_namespace_name`**(*namespace*, *name*)

Checks that the namespace is defined by an enumeration and that the name is a member

**`has_regex_namespace_name`**(*namespace*, *name*)

Checks that the namespace is defined as a regular expression and the name matches it

### 2.14.4 BEL Parser

**class** `pybel.parser.parse_bel.`**`BelParser`**(*graph*, *namespace_dict=None*, *annotation_dict=None*, *namespace_regex=None*, *annotation_regex=None*, *allow_naked_names=False*, *allow_nested=False*, *allow_unqualified_translocations=False*, *citation_clearing=True*, *no_identifier_validation=False*, *autostreamline=True*)

Build a parser backed by a given dictionary of namespaces

**Parameters**

- **`graph`** (`pybel.BELGraph`) – The BEL Graph to use to store the network
- **`namespace_dict`** (*`dict[str, dict[str, str]]`*) – A dictionary of {namespace: {name: encoding}}. Delegated to *`pybel.parser.parse_identifier.IdentifierParser`*
- **`annotation_dict`** (*`dict[str, set[str]]`*) – A dictionary of {annotation: set of values}. Delegated to `pybel.parser.ControlParser`
- **`namespace_regex`** (*`dict[str, str]`*) – A dictionary of {namespace: regular expression strings}. Delegated to *`pybel.parser.parse_identifier.IdentifierParser`*
- **`annotation_regex`** (*`dict[str, str]`*) – A dictionary of {annotation: regular expression strings}. Delegated to `pybel.parser.ControlParser`
- **`allow_naked_names`** (*`bool`*) – If true, turn off naked namespace failures. Delegated to *`pybel.parser.parse_identifier.IdentifierParser`*
- **`allow_nested`** (*`bool`*) – If true, turn off nested statement failures. Delegated to *`pybel.parser.parse_identifier.IdentifierParser`*
- **`allow_unqualified_translocations`** (*`bool`*) – If true, allow translocations without TO and FROM clauses.
- **`citation_clearing`** (*`bool`*) – Should `SET Citation` statements clear evidence and all annotations? Delegated to `pybel.parser.ControlParser`
- **`autostreamline`** (*`bool`*) – Should the parser be streamlined on instantiation?

**`pmod = None`**
2.2.1

**`variant = None`**
2.2.2

**`fragment = None`**
2.2.3

**`location = None`**
2.2.4

**`psub = None`**
DEPRECATED: 2.2.X Amino Acid Substitutions

**`gsub = None`**
DEPRECATED: 2.2.X Sequence Variations

**trunc = None**
DEPRECATED Truncated proteins

**gmod = None**
PyBEL BEL Specification variant

**fusion = None**
2.6.1

**general_abundance = None**
2.1.1

**gene = None**
2.1.4

**mirna = None**
2.1.5

**protein = None**
2.1.6

**rna = None**
2.1.7

**complex_singleton = None**
2.1.2

**composite_abundance = None**
2.1.3

**molecular_activity = None**
2.4.1

**biological_process = None**
2.3.1

**pathology = None**
2.3.2

**activity = None**
2.3.3

**translocation = None**
2.5.1

**degradation = None**
2.5.2

**reactants = None**
2.5.3

**increases_tag = None**
3.1.1

**directly_increases_tag = None**
3.1.2

**decreases_tag = None**
3.1.3

**directly_decreases_tag = None**
3.1.4

**`analogous_tag = None`**
3.5.1

**`causes_no_change_tag = None`**
3.1.6

**`regulates_tag = None`**
3.1.7

**`negative_correlation_tag = None`**
3.2.1

**`positive_correlation_tag = None`**
3.2.2

**`association_tag = None`**
3.2.3

**`orthologous_tag = None`**
3.3.1

**`is_a_tag = None`**
3.4.5

**`equivalent_tag = None`**
PyBEL Variants

**`rate_limit_tag = None`**
3.1.5

**`subprocess_of_tag = None`**
3.4.6

**`transcribed_tag = None`**
3.3.2

**`translated_tag = None`**
3.3.3

**`has_member_tag = None`**
3.4.1

**`abundance_list = None`**
3.4.2

**`biomarker_tag = None`**
3.5.2

**`prognostic_biomarker_tag = None`**
3.5.3

**`causal_relation_tags = None`**
3.1 Causal Relationships - nested. Not enabled by default.

**`namespace_dict`**
The dictionary of {namespace: {name: encoding}} stored in the internal identifier parser

**Return type** dict[str,dict[str,str]]

**`namespace_regex`**
The dictionary of {namespace keyword: compiled regular expression} stored the internal identifier parser

**Return type** dict[str,re]

**annotation_dict**
A dictionary of annotations to their set of values

**Return type** dict[str,set[str]]

**annotation_regex**
A dictionary of annotations defined by regular expressions {annotation keyword: string regular expression}

**Return type** dict[str,str]

**allow_naked_names**
Should naked names be parsed, or should errors be thrown?

**Return type** bool

**get_annotations**()
Get current annotations in this parser

**Return type** dict

**clear**()
Clears the graph and all control parser data (current citation, annotations, and statement group)

**handle_nested_relation**(*line*, *position*, *tokens*)
Handles nested statements. If `allow_nested` is False, raises a warning.

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** NestedRelationWarning

**check_function_semantics**(*line*, *position*, *tokens*)
Raises an exception if the function used on the tokens is wrong

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** InvalidFunctionSemantic

**handle_term**(*line*, *position*, *tokens*)
Handles BEL terms (the subject and object of BEL relations)

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**handle_has_members**(*line*, *position*, *tokens*)
Handles list relations like `p(X) hasMembers list(p(Y), p(Z), ...)`

**Parameters**

- **line** (*str*) – The line being parsed

- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**handle_has_components**(*line*, *position*, *tokens*)
Handles list relations like `p(X) hasComponents list(p(Y), p(Z), ...)`

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**handle_unqualified_relation**(*line*, *position*, *tokens*)
Handles unqualified relations

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**handle_label_relation**(*line*, *position*, *tokens*)
Handles statements like `p(X) label "Label for X"`

**Parameters**

- **line** (*str*) – The line being parsed
- **position** (*int*) – The position in the line being parsed
- **tokens** (*pyparsing.ParseResult*) – The tokens from PyParsing

**Raises** RelabelWarning

**ensure_node**(*tokens*)
Turns parsed tokens into canonical node name and makes sure its in the graph

**Parameters tokens** (*pyparsing.ParseResult*) – Tokens from PyParsing

**Returns** A pair of the PyBEL node tuple and the PyBEL node data dictionary

**Return type** tuple[tuple, dict]

### 2.14.5 Sub-Parsers

**class** pybel.parser.modifiers.**FusionParser**(*identifier_parser=None*)
Parses the BEL representation of gene and gene product fusions

**Parameters identifier_parser** (IdentifierParser) – An identifier parser for checking the 3P and 5P partners

**class** pybel.parser.modifiers.**TruncationParser**
Parses a protein trunctation and normalizes to HGVS

## 2.15 Utilities

Some utilities that are used throughout the software are explained here:

### 2.15.1 General Utilities

`pybel.utils.`**`expand_dict`**`(`*flat_dict*, *sep='_'*`)`

Expands a flattened dictionary

**Parameters**

- **flat_dict** (*dict*) – a nested dictionary that has been flattened so the keys are composite
- **sep** (*str*) – the separator between concatenated keys

**Return type** dict

`pybel.utils.`**`flatten_dict`**`(`*d*, *parent_key=''*, *sep='_'*`)`

Flattens a nested dictionary.

**Parameters**

- **d** (*dict or MutableMapping*) – A nested dictionary
- **parent_key** (*str*) – The parent's key. This is a value for tail recursion, so don't set it yourself.
- **sep** (*str*) – The separator used between dictionary levels

**Return type** dict

**See also:**

http://stackoverflow.com/a/6027615

`pybel.utils.`**`flatten_graph_data`**`(`*graph*`)`

Returns a new graph with flattened edge data dictionaries.

**Parameters** **graph** (*nx.MultiDiGraph*) – A graph with nested edge data dictionaries

**Returns** A graph with flattened edge data dictionaries

**Return type** nx.MultiDiGraph

`pybel.utils.`**`list2tuple`**`(`*l*`)`

Recursively converts a nested list to a nested tuple

**Return type** tuple

`pybel.utils.`**`get_version`**`()`

Gets the current PyBEL version

**Returns** The current PyBEL version

**Return type** str

`pybel.utils.`**`tokenize_version`**`(`*version_string*`)`

Tokenizes a version string to a tuple. Truncates qualifiers like `-dev`.

**Parameters** **version_string** (*str*) – A version string

**Returns** A tuple representing the version string

**Return type** tuple

```
>>> tokenize_version('0.1.2-dev')
(0, 1, 2)
```

`pybel.utils.`**`citation_dict_to_tuple`**`(`*citation*`)`

Convert the `d[CITATION]` entry in an edge data dictionary to a tuple

**Parameters** **citation** (*dict*) –

**Return type** tuple[str]

pybel.utils.**flatten_citation**(*citation*)
Flattens a citation dict, from the `d[CITATION]` entry in an edge data dictionary

**Parameters** **citation** (*dict[str, str]*) – A PyBEL citation data dictionary

**Return type** str

pybel.utils.**ensure_quotes**(*s*)
Quote a string that isn't solely alphanumeric

**Return type** str

pybel.utils.**valid_date**(*s*)
Checks that a string represents a valid date in ISO 8601 format YYYY-MM-DD

**Return type** bool

pybel.utils.**valid_date_version**(*s*)
Checks that the string is a valid date versions string

**Return type** bool

pybel.utils.**parse_datetime**(*s*)
Tries to parse a datetime object from a standard datetime format or date format

**Parameters** **s** (*str*) – A string representing a date or datetime

**Returns** A parsed date object

**Return type** datetime.date

pybel.utils.**hash_node**(*node_tuple*)
Converts a PyBEL node tuple to a hash

**Parameters** **node_tuple** (*tuple*) – A BEL node

**Returns** A hashed version of the node tuple using `hashlib.sha512()` hash of the binary pickle dump

**Return type** str

pybel.utils.**hash_edge**(*u*, *v*, *data*)
Converts an edge tuple to a hash

**Parameters**

- **u** (*tuple*) – The source BEL node
- **v** (*tuple*) – The target BEL node
- **data** (*dict*) – The edge's data dictionary

**Returns** A hashed version of the edge tuple using md5 hash of the binary pickle dump of u, v, and the json dump of d

**Return type** str

pybel.utils.**subdict_matches**(*target*, *query*, *partial_match=True*)
Checks if all the keys in the query dict are in the target dict, and that their values match

1. Checks that all keys in the query dict are in the target dict
2. **Matches the values of the keys in the query dict**

(a) If the value is a string, then must match exactly

(b) If the value is a set/list/tuple, then will match any of them

(c) If the value is a dict, then recursively check if that subdict matches

**Parameters**

- **`target`** (*`dict`*) – The dictionary to search
- **`query`** (*`dict`*) – A query dict with keys to match
- **`partial_match`** (*`bool`*) – Should the query values be used as partial or exact matches? Defaults to `True`.

**Returns** if all keys in b are in target_dict and their values match

**Return type** bool

`pybel.utils.`**`hash_dump`**`(`*data*`)`

Hashes an arbitrary JSON dictionary by dumping it in sorted order, encoding it in UTF-8, then hashing the bytes

**Parameters** **`data`** (*`dict or list or tuple`*) – An arbitrary JSON-serializable object

**Return type** str

`pybel.utils.`**`hash_citation`**`(`*type*, *reference*`)`

Creates a hash for a type/reference pair of a citation

**Parameters**

- **`type`** (*`str`*) – The corresponding citation type
- **`reference`** (*`str`*) – The citation reference

**Return type** str

`pybel.utils.`**`hash_evidence`**`(`*text*, *type*, *reference*`)`

Creates a hash for an evidence and its citation

**Parameters**

- **`text`** (*`str`*) – The evidence text
- **`type`** (*`str`*) – The corresponding citation type
- **`reference`** (*`str`*) – The citation reference

**Return type** str

## 2.15.2 IO Utilities

`pybel.io.line_utils.`**`parse_lines`**`(`*graph*, *lines*, *manager=None*, *allow_nested=False*, *citation_clearing=True*, ***kwargs*`)`

Parses an iterable of lines into this graph. Delegates to `parse_document()`, `parse_definitions()`, and `parse_statements()`.

**Parameters**

- **`graph`** (`BELGraph`) – A BEL graph
- **`lines`** (*`iter[str]`*) – An iterable over lines of BEL script
- **`manager`** (*`None or str or Manager`*) – An RFC-1738 database connection string, a pre-built `Manager`, or `None` for default connection

- **allow_nested** (*bool*) – If true, turns off nested statement failures
- **citation_clearing** (*bool*) – Should `SET Citation` statements clear evidence and all annotations? Delegated to `pybel.parser.ControlParser`

**Warning:** These options allow concessions for parsing BEL that is either **WRONG** or **UNSCIENTIFIC**. Use them at risk to reproducibility and validity of your results.

**Parameters**

- **allow_naked_names** (*bool*) – If true, turns off naked namespace failures
- **allow_unqualified_translocations** (*bool*) – If true, allow translocations without TO and FROM clauses.
- **no_identifier_validation** (*bool*) – If true, turns off namespace validation

### 2.15.3 Parser Utilities

`pybel.parser.utils.`**`is_int`**`(`*s*`)`

Determines if an object can be cast to an int

**Parameters** **s** – any object

**Returns** true if argument can be cast to an int:

**Return type** bool

`pybel.parser.utils.`**`nest`**`(`**content*`)`

Defines a delimited list by enumerating each element of the list

`pybel.parser.utils.`**`one_of_tags`**`(`*tags*, *canonical_tag*, *name=None*`)`

This is a convenience method for defining the tags usable in the `BelParser`. For example, statements like g(HGNC:SNCA) can be expressed also as geneAbundance(HGNC:SNCA). The language must define multiple different tags that get normalized to the same thing.

**Parameters**

- **tags** (*list[str]*) – a list of strings that are the tags for a function. For example, ['g', 'geneAbundance'] for the abundance of a gene
- **canonical_tag** (*str*) – the preferred tag name. Does not have to be one of the tags. For example, 'GeneAbundance' (note capitalization) is used for the abundance of a gene
- **name** (*str*) – this is the key under which the value for this tag is put in the PyParsing framework.

**Return type** `pyparsing.ParseElement`

`pybel.parser.utils.`**`triple`**`(`*subject*, *relation*, *obj*`)`

Builds a simple triple in PyParsing that has a `subject relation object` format

### 2.15.4 Canonicalization Utilities

This module helps handle node data dictionaries

`pybel.tokens.`**`hash_node_dict`**`(`*node_dict*`)`

Hashes a PyBEL node data dictionary

**Parameters** **node_dict** (*dict*) –

**Return type** str

`pybel.tokens.`**`node_to_tuple`**(*tokens*)

Given tokens from either PyParsing, or following the PyBEL node data dictionary model, create a PyBEL node tuple.

**Parameters** **tokens** (*ParseObject or dict*) – Either a PyParsing ParseObject or a PyBEL node data dictionary

**Return type** tuple

`pybel.tokens.`**`sort_dict_list`**(*tokens*)

Sorts a list of PyBEL data dictionaries to their canonical ordering

## 2.16 Domain Specific Language

PyBEL has a partially implemented domain specific language that makes it much easier to programmatically create and populate *pybel.BELGraph* instances.

**class** `pybel.dsl.`**`abundance`**(*namespace*, *name=None*, *identifier=None*)

Builds an abundance node data dictionary

**Parameters**

- **namespace** (*str*) – The name of the database used to identify this entity
- **name** (*str*) – The database's preferred name or label for this entity
- **identifier** (*str*) – The database's identifier for this entity

Example:

```
>>> bioprocess(namespace='CHEBI', name='water')
```

**class** `pybel.dsl.`**`gene`**(*namespace*, *name=None*, *identifier=None*, *variants=None*)

Builds a gene node data dictionary

**Parameters**

- **namespace** (*str*) – The name of the database used to identify this entity
- **name** (*str*) – The database's preferred name or label for this entity
- **identifier** (*str*) – The database's identifier for this entity
- **variants** (*list[Variant]*) – A list of variants

**class** `pybel.dsl.`**`rna`**(*namespace*, *name=None*, *identifier=None*, *variants=None*)

Builds an RNA node data dictionary

**Parameters**

- **namespace** (*str*) – The name of the database used to identify this entity
- **name** (*str*) – The database's preferred name or label for this entity
- **identifier** (*str*) – The database's identifier for this entity
- **variants** (*list[Variant]*) – A list of variants

Example: AKT1 protein coding gene's RNA:

```
>>> rna(namespace='HGNC', name='AKT1', identifier='391')
```

Non-coding RNA's can also be encoded such as U85:

```
>>> rna(namespace='SNORNABASE', identifer='SR0000073')
```

**class** `pybel.dsl.`**`mirna`**(*namespace*, *name=None*, *identifier=None*, *variants=None*)

Builds a miRNA node data dictionary

**Parameters**

- **namespace** (*str*) – The name of the database used to identify this entity
- **name** (*str*) – The database's preferred name or label for this entity
- **identifier** (*str*) – The database's identifier for this entity
- **variants** (*list[Variant]*) – A list of variants

Human miRNA's are listed on HUGO's MicroRNAs (MIR) gene family.

MIR1-1 from HGNC:

```
>>> mirna(namespace='HGNC', name='MIR1-1', identifier='31499')
```

MIR1-1 from miRBase:

```
>>> mirna(namespace='MIRBASE', identifier='MI0000651')
```

MIR1-1 from Entrez Gene

```
>>> mirna(namespace='ENTREZ', identifier='406904')
```

**class** `pybel.dsl.`**`protein`**(*namespace*, *name=None*, *identifier=None*, *variants=None*)

Builds a protein node data dictionary

Returns the node data dictionary for a protein

**Parameters**

- **namespace** (*str*) – The name of the database used to identify this entity
- **name** (*str*) – The database's preferred name or label for this entity
- **identifier** (*str*) – The database's identifier for this entity
- **variants** (*list[Variant]*) – A list of variants

Example: AKT

```
>>> protein(namespace='HGNC', name='AKT1')
```

Example: AKT with optionally included HGNC database identifier

```
>>> protein(namespace='HGNC', name='AKT1', identifier='391')
```

Example: AKT with phosphorylation

```
>>> protein(namespace='HGNC', name='AKT', variants=[pmod('Ph', code='Thr',␣
↪position=308)])
```

**class** pybel.dsl.**complex_abundance**(*members*, *namespace=None*, *name=None*, *identifier=None*)

Builds a complex abundance node data dictionary with the optional ability to specificy a name

**Parameters**

- **members** (*list[BaseAbundance]*) – A list of PyBEL node data dictionaries
- **namespace** (*Optional[str]*) – The namespace from which the name originates
- **name** (*Optional[str]*) – The name of the complex
- **identifier** (*Optional[str]*) – The identifier in the namespace in which the name originates

**class** pybel.dsl.**composite_abundance**(*members*)

Builds a composite abundance node data dictionary

**Parameters** **members** (*list[BaseAbundance]*) – A list of PyBEL node data dictionaries

**class** pybel.dsl.**bioprocess**(*namespace*, *name=None*, *identifier=None*)

Builds a biological process node data dictionary

**Parameters**

- **namespace** (*str*) – The name of the database used to identify this entity
- **name** (*str*) – The database's preferred name or label for this entity
- **identifier** (*str*) – The database's identifier for this entity

Example:

```
>>> bioprocess(namespace='GO', name='apoptosis')
```

**class** pybel.dsl.**pathology**(*namespace*, *name=None*, *identifier=None*)

Builds a pathology node data dictionary

**Parameters**

- **namespace** (*str*) – The name of the database used to identify this entity
- **name** (*str*) – The database's preferred name or label for this entity
- **identifier** (*str*) – The database's identifier for this entity

Example:

```
>>> pathology(namespace='DO', name='Alzheimer Disease')
```

**class** pybel.dsl.**reaction**(*reactants*, *products*)

Builds a reaction node data dictionary

**Parameters**

- **reactants** (*list[BaseAbundance]*) – A list of PyBEL node data dictionaries representing the reactants
- **products** (*list[BaseAbundance]*) – A list of PyBEL node data dictionaries representing the products

Example:

```
>>> reaction([protein(namespace='HGNC', name='KNG1')], [abundance(namespace='CHEBI
↪', name='bradykinin')])
```

**class** pybel.dsl.**pmod**(*name*, *code=None*, *position=None*, *namespace=None*, *identifier=None*)

Builds a protein modification variant dictionary

**Parameters**

- **name** (*str*) – The name of the modification
- **code** (*str*) – The three letter amino acid code for the affected residue. Capital first letter.
- **position** (*int*) – The position of the affected residue
- **namespace** (*str*) – The namespace to which the name of this modification belongs
- **identifier** (*str*) – The identifier of the name of the modification

Either the name or the identifier must be used. If the namespace is omitted, it is assumed that a name is specified from the BEL default namespace.

Example from BEL default namespace:

```
>>> pmod('Ph', code='Thr', position=308)
```

Example from custom namespace:

```
>>> pmod(name='protein phosphorylation', namespace='GO', code='Thr', position=308)
```

Example from custom namespace additionally qualified with identifier:

```
>>> pmod(name='protein phosphorylation', namespace='GO', identifier='GO:0006468',␣
↪code='Thr', position=308)
```

**class** pybel.dsl.**gmod**(*name*, *namespace=None*, *identifier=None*)

Builds a gene modification variant dictionary

**Parameters**

- **name** (*str*) – The name of the gene modification
- **namespace** (*Optional[str]*) – The namespace of the gene modification
- **identifier** (*Optional[str]*) – The identifier of the name in the database

Either the name or the identifier must be used. If the namespace is omitted, it is assumed that a name is specified from the BEL default namespace.

Example from BEL default namespace:

```
>>> gmod(name='Me')
```

Example from custom namespace:

```
>>> gmod(name='DNA methylation', namespace='GO', identifier='GO:0006306',)
```

**class** pybel.dsl.**hgvs**(*variant*)

Builds a HGVS variant dictionary

**Parameters** **variant** (*str*) – The HGVS variant string

Example:

```
>>> protein(namespace='HGNC', name='AKT1', variants=[hgvs('p.Ala127Tyr')])
```

**class** pybel.dsl.**protein_substitution**(*from_aa*, *position*, *to_aa*)

Builds a HGVS variant dictionary for the given protein substitution

**Parameters**

- **from_aa** (*str*) – The 3-letter amino acid code of the original residue
- **position** (*int*) – The position of the residue
- **to_aa** (*str*) – The 3-letter amino acid code of the new residue

Example:

```
>>> protein(namespace='HGNC', name='AKT1', variants=[protein_substitution('Ala',␣
↪127, 'Tyr')])
```

**class** pybel.dsl.**fusion_range**(*reference, start, stop*)

Creates a fusion range data dictionary

**Parameters**

- **reference** (*str*) – The reference code
- **or str start** (*int*) – The start position, either specified by its integer position, or '?'
- **or str stop** (*int*) – The stop position, either specified by its integer position, '?', or '*

Example fully specified RNA fusion range:

```
>>> fusion_range('r', 1, 79)
```

**as_tuple**()

**Return type** tuple

**class** pybel.dsl.**missing_fusion_range**

Builds a missing fusion range data dictionary

**as_tuple**()

**Return type** tuple

**class** pybel.dsl.**protein_fusion**(*partner_5p, partner_3p, range_5p=None, range_3p=None*)

Builds a protein fusion data dictionary

**Parameters**

- **partner_5p** (pybel.dsl.protein) – A PyBEL node data dictionary for the 5-prime partner
- **partner_3p** (pybel.dsl.protein) – A PyBEL node data dictionary for the 3-prime partner
- **range_5p** (*Optional[FusionRangeBase]*) – A fusion range for the 5-prime partner
- **range_3p** (*Optional[FusionRangeBase]*) – A fusion range for the 3-prime partner

**class** pybel.dsl.**rna_fusion**(*partner_5p, partner_3p, range_5p=None, range_3p=None*)

Builds an RNA fusion data dictionary

**Parameters**

- **partner_5p** (pybel.dsl.rna) – A PyBEL node data dictionary for the 5-prime partner
- **partner_3p** (pybel.dsl.rna) – A PyBEL node data dictionary for the 3-prime partner
- **range_5p** (*Optional[FusionRangeBase]*) – A fusion range for the 5-prime partner

- **range_3p** (*Optional[FusionRangeBase]*) – A fusion range for the 3-prime partner

Example, with fusion ranges using the 'r' qualifier:

```
>>> rna_fusion(
>>> ... partner_5p=rna(namespace='HGNC', name='TMPRSS2'),
>>> ... range_5p=fusion_range('r', 1, 79),
>>> ... partner_3p=rna(namespace='HGNC', name='ERG'),
>>> ... range_3p=fusion_range('r', 312, 5034)
>>> )
```

Example with missing fusion ranges:

```
>>> rna_fusion(
>>> ... partner_5p=rna(namespace='HGNC', name='TMPRSS2'),
>>> ... partner_3p=rna(namespace='HGNC', name='ERG'),
>>> )
```

**class** pybel.dsl.**gene_fusion**(*partner_5p*, *partner_3p*, *range_5p=None*, *range_3p=None*)

Builds a gene fusion data dictionary

**Parameters**

- **partner_5p** (pybel.dsl.gene) – A PyBEL node data dictionary for the 5-prime partner
- **partner_3p** (pybel.dsl.gene) – A PyBEL node data dictionary for the 3-prime partner
- **range_5p** (*Optional[FusionRangeBase]*) – A fusion range for the 5-prime partner
- **range_3p** (*Optional[FusionRangeBase]*) – A fusion range for the 3-prime partner

Example, using fusion ranges with the 'c' qualifier

```
>>> gene_fusion(
>>> ... partner_5p=gene(namespace='HGNC', name='TMPRSS2'),
>>> ... range_5p=fusion_range('c', 1, 79),
>>> ... partner_3p=gene(namespace='HGNC', name='ERG'),
>>> ... range_3p=fusion_range('c', 312, 5034)
>>> )
```

Example with missing fusion ranges:

```
>>> gene_fusion(
>>> ... partner_5p=gene(namespace='HGNC', name='TMPRSS2'),
>>> ... partner_3p=gene(namespace='HGNC', name='ERG'),
>>> )
```

pybel.dsl.**activity**(*name=None*, *namespace=None*, *identifier=None*, *location=None*)

Makes a subject/object modifier dictionary

**Parameters**

- **name** (*str*) – The name of the activity. If no namespace given, uses BEL default namespace
- **namespace** (*Optional[str]*) – The namespace of the activity
- **identifier** (*Optional[str]*) – The identifier of the name in the database
- **location** (*Optional[dict]*) – An entity from `pybel.dsl.entity()` representing the location of the node

**Return type** dict

`pybel.dsl.degradation(location=None)`
Adds the degradation

**Parameters** **location** (*Optional[dict]*) – An entity from `pybel.dsl.entity()` representing the location of the node

**Return type** dict

`pybel.dsl.translocation(from_loc, to_loc)`
Makes a translocation dict

**Parameters**

- **from_loc** (*dict*) – An entity dictionary from `pybel.dsl.entity()`
- **to_loc** (*dict*) – An entity dictionary from `pybel.dsl.entity()`

**Return type** dict

`pybel.dsl.secretion()`
Convenient wrapper representing the `translocation()` from the intracellular location to the extracellular space

**Return type** dict

`pybel.dsl.cell_surface_expression()`
Convenient wrapper representing the `translocation()` from the intracellular location to the cell surface

**Return type** dict

`exception pybel.dsl.PyBELDSLException`
Raised when problems with the DSL

## 2.17 Logging Messages

### 2.17.1 Errors

This module contains base exceptions that are shared through the package

`exception pybel.exceptions.PyBelWarning`
The base class for warnings during compilation from which PyBEL can recover

`exception pybel.exceptions.PyBELCanonicalizeError`
Raised when problem canonicalizing a node

### 2.17.2 Parse Exceptions

A message for "General Parser Failure" is displayed when a problem was caused due to an unforseen error. The line number and original statement are printed for the user to debug.

`exception pybel.parser.exc.PyBelParserWarning(line_number, line, position, *args)`
Base PyBEL parser exception, which holds the line and position where a parsing problem occurred

**Parameters**

- **line_number** (*int*) – The line number on which this warning occurred
- **line** (*str*) – The content of the line

- **position** (*int*) – The position within the line where the warning occurred
- **args** – Additional arguments to supply to the super class

**exception** `pybel.parser.exc.`**`BelSyntaxError`**(*line_number*, *line*, *position*, **args*)

For general syntax errors

**Parameters**

- **line_number** (*int*) – The line number on which this warning occurred
- **line** (*str*) – The content of the line
- **position** (*int*) – The position within the line where the warning occurred
- **args** – Additional arguments to supply to the super class

**exception** `pybel.parser.exc.`**`InconsistentDefinitionError`**(*line_number*, *line*, *position*, *definition*)

Base PyBEL error for redefinition

**exception** `pybel.parser.exc.`**`RedefinedNamespaceError`**(*line_number*, *line*, *position*, *definition*)

Raised when a namespace is redefined

**exception** `pybel.parser.exc.`**`RedefinedAnnotationError`**(*line_number*, *line*, *position*, *definition*)

Raised when an annotation is redefined

**exception** `pybel.parser.exc.`**`NameWarning`**(*line_number*, *line*, *position*, *name*, **args*)

The base class for errors related to nomenclature

**exception** `pybel.parser.exc.`**`NakedNameWarning`**(*line_number*, *line*, *position*, *name*, **args*)

Raised when there is an identifier without a namespace. Enable lenient mode to suppress

**exception** `pybel.parser.exc.`**`MissingDefaultNameWarning`**(*line_number*, *line*, *position*, *name*, **args*)

Raised if reference to value not in default namespace

**exception** `pybel.parser.exc.`**`NamespaceIdentifierWarning`**(*line_number*, *line*, *position*, *namespace*, *name*)

The base class for warnings related to namespace:name identifiers

**Parameters**

- **line_number** (*int*) – The line number of the line that caused the exception
- **line** (*str*) – The line that caused the exception
- **position** (*int*) – The line's position of the exception
- **namespace** (*str*) – The namespace of the identifier
- **name** (*str*) – The name of the identifier

**exception** `pybel.parser.exc.`**`UndefinedNamespaceWarning`**(*line_number*, *line*, *position*, *namespace*, *name*)

Raised if reference made to undefined namespace

**Parameters**

- **line_number** (*int*) – The line number of the line that caused the exception
- **line** (*str*) – The line that caused the exception
- **position** (*int*) – The line's position of the exception
- **namespace** (*str*) – The namespace of the identifier

- **name** (*str*) – The name of the identifier

**exception** pybel.parser.exc.**MissingNamespaceNameWarning**(*line_number*, *line*, *position*, *namespace*, *name*)

Raised if reference to value not in namespace

**Parameters**

- **line_number** (*int*) – The line number of the line that caused the exception
- **line** (*str*) – The line that caused the exception
- **position** (*int*) – The line's position of the exception
- **namespace** (*str*) – The namespace of the identifier
- **name** (*str*) – The name of the identifier

**exception** pybel.parser.exc.**MissingNamespaceRegexWarning**(*line_number*, *line*, *position*, *namespace*, *name*)

Raised if reference not matching regex

**Parameters**

- **line_number** (*int*) – The line number of the line that caused the exception
- **line** (*str*) – The line that caused the exception
- **position** (*int*) – The line's position of the exception
- **namespace** (*str*) – The namespace of the identifier
- **name** (*str*) – The name of the identifier

**exception** pybel.parser.exc.**AnnotationWarning**(*line_number*, *line*, *position*, *annotation*, **args*)

Base exception for annotation warnings

**exception** pybel.parser.exc.**UndefinedAnnotationWarning**(*line_number*, *line*, *position*, *annotation*, **args*)

Raised when an undefined annotation is used

**exception** pybel.parser.exc.**MissingAnnotationKeyWarning**(*line_number*, *line*, *position*, *annotation*, **args*)

Raised when trying to unset an annotation that is not set

**exception** pybel.parser.exc.**AnnotationIdentifierWarning**(*line_number*, *line*, *position*, *annotation*, *value*)

Base exception for annotation:value pairs

**exception** pybel.parser.exc.**IllegalAnnotationValueWarning**(*line_number*, *line*, *position*, *annotation*, *value*)

Raised when an annotation has a value that does not belong to the original set of valid annotation values.

**exception** pybel.parser.exc.**MissingAnnotationRegexWarning**(*line_number*, *line*, *position*, *annotation*, *value*)

Raised if annotation doesn't match regex

**exception** pybel.parser.exc.**VersionFormatWarning**(*line_number*, *line*, *position*, *version_string*)

Raised if the version string doesn't adhere to semantic versioning or `YYYYMMDD` format

**exception** pybel.parser.exc.**MetadataException**(*line_number*, *line*, **args*)

Base exception for issues with document metadata

**exception** pybel.parser.exc.**MalformedMetadataException**(*line_number*, *line*, **args*)

Raised when an invalid metadata line is encountered

**exception** `pybel.parser.exc.`**`InvalidMetadataException`**(*line_number*, *line*, *position*, *key*, *value*)

Raised when an incorrect document metadata key is used. Valid document metadata keys are:

- `Authors`
- `ContactInfo`
- `Copyright`
- `Description`
- `Disclaimer`
- `Licenses`
- `Name`
- `Version`

**See also:**

BEL specification on the properties section

**exception** `pybel.parser.exc.`**`MissingMetadataException`**(*key*)

Raised when a BEL Script is missing critical metadata.

**exception** `pybel.parser.exc.`**`InvalidCitationLengthException`**(*line_number*, *line*, *position*, **args*)

Base exception raised when the format for a citation is wrong.

**Parameters**

- **line_number** (*int*) – The line number on which this warning occurred
- **line** (*str*) – The content of the line
- **position** (*int*) – The position within the line where the warning occurred
- **args** – Additional arguments to supply to the super class

**exception** `pybel.parser.exc.`**`CitationTooShortException`**(*line_number*, *line*, *position*, **args*)

Raised when a citation does not have the minimum of {type, name, reference}.

**Parameters**

- **line_number** (*int*) – The line number on which this warning occurred
- **line** (*str*) – The content of the line
- **position** (*int*) – The position within the line where the warning occurred
- **args** – Additional arguments to supply to the super class

**exception** `pybel.parser.exc.`**`CitationTooLongException`**(*line_number*, *line*, *position*, **args*)

Raised when a citation has more than the allowed entries, {type, name, reference, date, authors, comments}.

**Parameters**

- **line_number** (*int*) – The line number on which this warning occurred
- **line** (*str*) – The content of the line
- **position** (*int*) – The position within the line where the warning occurred
- **args** – Additional arguments to supply to the super class

**exception** pybel.parser.exc.**MissingCitationException**(*line_number*, *line*, *position*, **args*)

Raised when trying to parse a BEL statement, but no citation is currently set. This might be due to a previous error in the formatting of a citation.

Though it's not a best practice, some BEL curators set other annotations before the citation. If this is the case in your BEL document, and you're absolutly sure that all UNSET statements are correctly written, you can use `citation_clearing=True` as a keyword argument in any of the IO functions in `pybel.from_lines()`, `pybel.from_url()`, or `pybel.from_path()`.

**Parameters**

- **line_number** (*int*) – The line number on which this warning occurred
- **line** (*str*) – The content of the line
- **position** (*int*) – The position within the line where the warning occurred
- **args** – Additional arguments to supply to the super class

**exception** pybel.parser.exc.**MissingSupportWarning**(*line_number*, *line*, *position*, **args*)

Raised when trying to parse a BEL statement, but no evidence is currently set. All BEL statements must be qualified with evidence.

If your data is serialized from a database and provenance information is not readily accessible, consider referencing the publication for the database, or a url pointing to the data from either a programmatically or human-readable endpoint.

**Parameters**

- **line_number** (*int*) – The line number on which this warning occurred
- **line** (*str*) – The content of the line
- **position** (*int*) – The position within the line where the warning occurred
- **args** – Additional arguments to supply to the super class

**exception** pybel.parser.exc.**InvalidCitationType**(*line_number*, *line*, *position*, *citation_type*)

Raise when a citation is set with an incorrect type. Valid citation types include:

- `Book`
- `PubMed`
- `Journal`
- `Online Resource`
- `URL`
- `DOI`
- `Other`

**See also:**

OpenBEL wiki on citations

**exception** pybel.parser.exc.**InvalidPubMedIdentifierWarning**(*line_number*, *line*, *position*, *reference*)

Raised when a citation is set whose type is `PubMed` but whose database identifier is not a valid integer.

**exception** pybel.parser.exc.**MalformedTranslocationWarning**(*line_number*, *line*, *position*, *tokens*)

Raised when there is a translocation statement without location information.

**exception** pybel.parser.exc.**PlaceholderAminoAcidWarning**(*line_number*, *line*, *position*, *code*)

Raised when an invalid amino acid code is given.

One example might be the usage of X, which is a colloquial signifier for a truncation in a given position. Text mining efforts for knowledge extraction make this mistake often. X might also signify a placeholder amino acid.

**exception** pybel.parser.exc.**NestedRelationWarning**(*line_number*, *line*, *position*, **args*)

Raised when encountering a nested statement. See our the docs for an explanation of why we explicitly do not support nested statements.

**Parameters**

- **line_number** (*int*) – The line number on which this warning occurred
- **line** (*str*) – The content of the line
- **position** (*int*) – The position within the line where the warning occurred
- **args** – Additional arguments to supply to the super class

**exception** pybel.parser.exc.**LexicographyWarning**

Raised when encountering improper capitalization of namespace/annotation names.

**exception** pybel.parser.exc.**InvalidFunctionSemantic**(*line_number*, *line*, *position*, *function*, *namespace*, *name*, *allowed_functions*)

Raised when an invalid function is used for a given node.

For example, an HGNC symbol for a protein-coding gene YFG cannot be referenced as an miRNA with `m(HGNC:YFG)`

**exception** pybel.parser.exc.**RelabelWarning**(*line_number*, *line*, *position*, *node*, *old_label*, *new_label*)

Raised when a node is relabeled

## 2.18 Extensions

Extensions for PyBEL can be imported from the `pybel.ext` namespace just like normal modules and packages:

```
import pybel.ext.extension
# or
from pybel.ext import extension as ex
# or
from pybel.ext.extension import a_function as af
# or
from pybel.ext.extension import *
# or, even
from pybel.ext import *
```

This magic is brought to you by import hooks and `pkg_resources`.

To create your own extension, simply register a `pybel.ext` entry point in your package's `setup.py`:

```
setuptools.setup(
    ...
    entry_points = {
        'pybel.ext':
            ['name_of_your_extension = package.module']
    }
    ...
)
```

This works just like the standard `console_scripts` entry point and the syntax follows all the same rules.

Your extension will then be importable as `pybel.ext.name_of_your_extension`

**Warning:** PyBEL does not check for collisions in extension names. Please be careful when naming your extension!

See the test extension on GitHub to see a working example of an extension.

`pybel.ext` exists as a package on its own to trick the Python import system. . . if we just did this work in PyBEL's `__init__.py`, there could be trouble.

## 2.19 Roadmap

This project road map documents not only the PyBEL repository, but the PyBEL Tools and PyBEL Web repositories as well as the Bio2BEL project.

### 2.19.1 PyBEL

- **Performance improvements**
  - Parallelization of parsing
  - On-the-fly validation with OLS or MIRIAM

### 2.19.2 Bio2BEL

- **Generation of new namespaces, equivalencies, and hierarchical knowledge (isA and partOf relations)**
  - FlyBase
  - InterPro
  - UniProt
  - ChEBML
  - Human Phenotype Ontology
  - Uber Anatomy Ontology
  - HGNC Gene Families
  - Enyzme Classification
- **Integration of knowledge sources**
  - ChEMBL
  - Comparative Toxicogenomics Database
  - BRENDA
  - MetaCyc

  - Protein complex definitions
- **Integration of analytical pipelines**
  - LD Block Analysis
  - Gene Co-expression Analysis
  - Differential Gene Expression Analysis

### 2.19.3 PyBEL Tools

- **Biological Grammar**
  - Network motif identification
  - Stability analysis
  - **Prior knowledge comparision**
    * Molecular activity annotation
    * SNP Impact
- **Implementation of standard BEL Algorithms**
  - RCR
  - NPA
  - SST
- **Development of new algorithms**
  - Heat diffusion algorithms
  - AETIONOMY Workflow 1 (Drug Repurposing)
  - Cart Before the Horse
- Metapath analysis
- Reasoning and inference rules
- Subgraph Expansion application in NeuroMMSigDB
- Chemical Enrichment in NeuroMMSigDB

### 2.19.4 PyBEL Web

- Integration with BELIEF
- Integration with NeuroMMSigDB
- Import and export from NDEx

## 2.20 Current Issues

### 2.20.1 Speed

**Speed is still an issue, because documents above 100K lines still take a couple minutes to run. This issue is** exacerbated by (optionally) logging output to the console, which can make it more than 3x or 4x as slow.

### 2.20.2 Namespaces

The default namespaces from OpenBEL do not follow a standard file format. They are similar to INI config files, but do not use consistent delimiters. Also, many of the namespaces don't respect that the delimiter should not be used in the namespace names. There are also lots of names with strange characters, which may have been caused by copying from a data source that had specfic escape characters without proper care.

### 2.20.3 Testing

Testing was very difficult because the example documents on the OpenBEL website had many semantic errors, such as using names and annotation values that were not defined within their respective namespace and annotation definition files. They also contained syntax errors like naked names, which are not only syntatically incorrect, but lead to bad science; and improper usage of activities, like illegally nesting an activity within a composite statement.

## 2.21 Technology

This page is meant to describe the development stack for PyBEL, and should be a useful introduction for contributors.

### 2.21.1 Versioning

PyBEL is versioned on GitHub so changes in its code can be tracked over time and to make use of the variety of software development plugins. Code is produced following the Git Flow philosophy, which means that new features are coded in branches off of the development branch and merged after they are triaged. Finally, develop is merged into master for releases. If there are bugs in releases that need to be fixed quickly, "hot fix" branches from master can be made, then merged back to master and develop after fixing the problem.

### 2.21.2 Testing in PyBEL

PyBEL is written with extensive unit testing and integration testing. Whenever possible, test- driven development is practiced. This means that new ideas for functions and features are encoded as blank classes/functions and directly writing tests for the desired output. After tests have been written that define how the code should work, the implementation can be written.

Test-driven development requires us to think about design before making quick and dirty implementations. This results in better code. Additionally, thorough testing suites make it possible to catch when changes break existing functionality.

Tests are written with the standard `unittest` library. Some functionality, such as the `mock` module, are only available as default in Python 3, so backports must be used for testing in Python 2

#### Unit Testing

Unit tests check that the functionality of the different parts of PyBEL work independently.

An example unit test can be found in `tests.test_parse_bel.TestAbundance.test_short_abundance`. It ensures that the parser is able to handle a given string describing the abundance of a chemical/other entity in BEL. It tests that the parser produces the correct output, that the BEL statement is converted to the correct internal representation. In this example, this is a tuple describing the abundance of oxygen atoms. Finally, it tests that this representation is added as a node in the underlying BEL graph with the appropriate attributes added.

### Integration Testing

Integration tests are more high level, and ensure that the software accomplishes more complicated goals by using many components. An example integration test is found in tests.test_import.TestImport.test_from_fileURL. This test ensures that a BEL script can be read and results in a NetworkX object that contains all of the information described in the script

### Tox

While IDEs like PyCharm provide excellent testing tools, they are not programmatic. Tox is python package that provides a CLI interface to run automated testing procedures (as well as other build functions, that aren't important to explain here). In PyBEL, it is used to run the unit tests in the `tests` folder with the `pytest` harness. It also runs `check-manifest`, builds the documentation with `sphinx`, and computes the code coverage of the tests. The entire procedure is defined in `tox.ini`. Tox also allows test to be done on many different versions of Python.

### Continuous Integration

Continuous integration is a philosophy of automatically testing code as it changes. PyBEL makes use of the Travis CI server to perform testing because of its tight integration with GitHub. Travis automatically installs git hooks inside GitHub so it knows when a new commit is made. Upon each commit, Travis downloads the newest commit from GitHub and runs the tests configured in the `.travis.yml` file in the top level of the PyBEL repository. This file effectively instructs the Travis CI server to run Tox. It also allows for the modification of the environment variables. This is used in PyBEL to test many different versions of python.

### Code Coverage

After building, Travis sends code coverage results to codecov.io. This site helps visualize untested code and track the improvement of testing coverage over time. It also integrates with GitHub to show which feature branches are inadequately tested. In development of PyBEL, inadequately tested code is not allowed to be merged into develop.

### Versioning

PyBEL uses semantic versioning. In general, the project's version string will has a suffix `-dev` like in `0.3.4-dev` throughout the development cycle. After code is merged from feature branches to develop and it is time to deploy, this suffix is removed and develop branch is merged into master.

The version string appears in multiple places throughout the project, so BumpVersion is used to automate the updating of these version strings. See .bumpversion.cfg for more information.

## 2.21.3 Deployment

PyBEL is also distributed through PyPI (pronounced Py-Pee-Eye). Travis CI has a wonderful integration with PyPI, so any time a tag is made on the master branch (and also assuming the tests pass), a new distribution is packed and sent to PyPI. Refer to the "deploy" section at the bottom of the `.travis.yml` file for more information, or the Travis CI PyPI deployment documentation. As a side note, Travis CI has an encryption tool so the password for the PyPI account can be displayed publicly on GitHub. Travis decrypts it before performing the upload to PyPI.

### Steps

1. `bumpversion release` on development branch
2. Push to git
3. After tests pass, merge develop in to master
4. After tests pass, create a tag on GitHub with the same name as the version number (on master)
5. Travis will automatically deploy to PyPI after tests pass. After checking deployment has been successful, switch to develop and `bumpversion patch`

# CHAPTER 3

# Indices and Tables

- genindex
- modindex
- search

# Python Module Index

# Index