INDRA Documentation

Release 1.21.0

B. M. Gyori, J. A. Bachman

Feb 01, 2022

CONTENTS

INDRA (the Integrated Network and Dynamical Reasoning Assembler) assembles information about causal mechanisms into a common format that can be used to build several different kinds of predictive and explanatory models. INDRA was originally developed for molecular systems biology and is currently being generalized to other domains.

In molecular biology, sources of mechanistic information include pathway databases, natural language descriptions of mechanisms by human curators, and findings extracted from the literature by text mining.

Mechanistic information from multiple sources is de-duplicated, standardized and assembled into sets of *Statements* with associated evidence. Sets of Statements can then be used to assemble both executable rule-based models (using PySB) and a variety of different types of network models.

CHAPTER

ONE

LICENSE AND FUNDING

INDRA is made available under the 2-clause BSD license. INDRA was developed with funding from ARO grant W911NF-14-1-0397, "Programmatic modelling for reasoning across complex mechanisms" under the DARPA Big Mechanism program, W911NF-14-1-0391, "Active context" under the DARPA Communicating with Computers program, "Global Reading and Assembly for Semantic Probabilisic World Models" in the DARPA World Modelers program, and the DARPA Automated Scientific Discovery Framework project.

CHAPTER

TWO

INSTALLATION

2.1 Installing Python

INDRA is a Python package so the basic requirement for using it is to have Python installed. Python is shipped with most Linux distributions and with OSX. INDRA works with Python 3.6 or higher.

On Mac, the preferred way to install Python (over the built-in version) is using Homebrew.

```
brew install python
```

On Windows, we recommend using Anaconda which contains compiled distributions of the scientific packages that INDRA depends on (numpy, scipy, pandas, etc).

2.2 Installing INDRA

2.2.1 Installing via Github

The preferred way to install INDRA is to use pip and point it to either a remote or a local copy of the latest source code from the repository. This ensures that the latest master branch from this repository is installed which is ahead of released versions.

To install directly from Github, do:

```
pip install git+https://github.com/sorgerlab/indra.git
```

Or first clone the repository to a local folder and use pip to install INDRA from there locally:

```
git clone https://github.com/sorgerlab/indra.git
cd indra
pip install .
```

2.2.2 Cloning the source code from Github

You may want to simply clone the source code without installing INDRA as a system-wide package.

```
git clone https://github.com/sorgerlab/indra.git
```

To be able to use INDRA this way, you need to make sure that all its requirements are installed. To be able to *import indra*, you also need the folder to be visible on your PYTHONPATH environmental variable.

2.2.3 Installing releases with pip

Releases of INDRA are also available via PyPI. You can install the latest released version of INDRA as

```
pip install indra
```

2.3 INDRA dependencies

INDRA depends on a few standard Python packages (e.g. rdflib, requests, objectpath). These packages are installed automatically by pip.

Below we provide a detailed description of some extra dependencies that may require special steps to install.

2.3.1 PySB and BioNetGen

INDRA builds on the PySB framework to assemble rule-based models of biochemical systems. The *pysb* python package is installed by the standard install procedure. However, to be able to generate mathematical model equations and to export to formats such as SBML, the BioNetGen framework also needs to be installed in a way that is visible to PySB. Detailed instructions are given in the PySB documentation.

2.3.2 Pyjnius

Pyjnius is currently not required for any of INDRA's features. However, to be able to use INDRA's optional JAR-based offline reading via the REACH and Eidos APIs, pyjnius is needed to allow using Java/Scala classes from Python.

1. Install JDK from Oracle: https://www.oracle.com/technetwork/java/javase/downloads/index.html. We recommend using Java 8 (INDRA is regularly tested with Java 8), however, Java 11 is also expected to be compatible, with possible extra configuration steps needed that are not described here.

4. Set JAVA_HOME to your JDK home directory, for instance

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home
```

3. Then first install cython followed by pyjnius (tested with version 1.1.4). These need to be broken up into two sequential calls to pip install.

```
pip install cython
pip install pyjnius==1.1.4
```

On Mac, you may need to install Legacy Java for OSX. If you have trouble installing it, you can try the following as an alternative. Edit

```
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Info.plist
```

(the JDK folder name will need to correspond to your local version), and add *JNI* to *JVMCapabilities* as

```
...
<dict>
    <key>JVMCapabilities</key>
    <array>
        <string>CommandLine</string>
        <string>JNI</string>
    </array>
...
```

2.3.3 Graphviz

Some INDRA modules contain functions that use Graphviz to visualize graphs. On most systems, doing

```
pip install pygraphviz
```

works. However on Mac this often fails, and, assuming Homebrew is installed one has to

```
brew install graphviz
pip install pygraphviz --install-option="--include-path=/usr/local/include/graphviz/" --
↪install-option="--library-path=/usr/local/lib/graphviz"
```

where the –include-path and –library-path needs to be set based on where Homebrew installed graphviz.

2.3.4 Optional additional dependencies

Some dependencies of INDRA are only needed by certain submodules or are only used in specialized use cases. These are not installed by default but are listed as "extra" requirements, and can be installed separately using pip. An extra dependency list (e.g. one called extra_list) can be installed as

```
pip install indra[extra_list]
```

You can also install all extra dependencies by doing

```
pip install indra --install-option="complete"
```

or

```
pip install indra[all]
```

In all of the above, you may replace *indra* with . (if you're in a local copy of the *indra* folder or with the Github URL of the INDRA repo, depending on your installation method. See also the corresponding pip documentation for more information.

The table below provides the name and the description of each "extra" list of dependencies.

Extra list name	Purpose
bel	BEL input processing and output assembly
trips_offline	Offline reading with local instance of TRIPS system
reach_offline	Offline reading with local instance of REACH system
eidos_offline	Offline reading with local instance of Eidos system
geneways	Genewayas reader input processing
sofia	SOFIA reader input processing
bbn	BBN reader input processing
sbml	SBML model export through the PySB Assembler
grounding	Packages for re-grounding and disambiguating entities
machine	Running a local instance of a "RAS machine"
explanation	Finding explanatory paths in rule-based models
aws	Accessing AWS compute and storage resources
graph	Assembling into a visualizing Graphviz graphs
plot	Create and display plots

2.4 Configuring INDRA

Various aspects of INDRA, including API keys, dependency locations, and Java memory limits, are parameterized by a configuration file that lives in ~/.config/indra/config.ini. The default configuration file is provided in indra/resources/default_config.ini, and is copied to ~/.config/indra/config.ini when INDRA starts if no configuration already exists. Every value in the configuration can also be set as an environment variable: for a given configuration key, INDRA will first check for an environment variable with that name and if not present, will use the value in the configuration file. In other words, an environment variable, when set, takes precedence over the value set in the config file.

Configuration values include:

- REACHPATH: The location of the JAR file containing a local instance of the REACH reading system
- EIDOSPATH: The location of the JAR file containing a local instance of the Eidos reading system
- SPARSERPATH: The location of a local instance of the Sparser reading system (path to a folder)
- DRUMPATH: The location of a local installation of the DRUM reading system (path to a folder)
- NDEX_USERNAME, NDEX_PASSWORD: Credentials for accessing the NDEx web service
- ELSEVIER_API_KEY, ELSEVIER_INST_KEY: Elsevier web service API keys
- BIOGRID_API_KEY: API key for BioGRID web service (see http://wiki.thebiogrid.org/doku.php/biogridrest)
- INDRA_DEFAULT_JAVA_MEM_LIMIT: Maximum memory limit for Java virtual machines launched by INDRA
- SITEMAPPER_CACHE_PATH: Path to an optional cache (a pickle file) for the SiteMapper's automatically obtained mappings.

CHAPTER

THREE

GETTING STARTED WITH INDRA

3.1 Importing INDRA and its modules

INDRA can be imported and used in a Python script or interactively in a Python shell. Note that similar to some other packages (e.g scipy), INDRA doesn't automatically import all its submodules, so *import indra* is not enough to access its submodules. Rather, one has to explicitly import each submodule that is needed. For example to access the BEL API, one has to

```
from indra.sources import bel
```

Similarly, each model output assembler has its own submodule under *indra.assemblers* with the assembler class accessible at the submodule level, so they can be imported as, for instance,

```
from indra.assemblers.pysb import PysbAssembler
```

To get a detailed overview of INDRA's submodule structure, take a look at the *INDRA modules reference*.

3.2 Basic usage examples

Here we show some basic usage examples of the submodules of INDRA. More complex usage examples are shown in the Tutorials section.

3.2.1 Reading a sentence with TRIPS

In this example, we read a sentence via INDRA's TRIPS submodule to produce an INDRA Statement.

```
from indra.sources import trips
sentence = 'MAP2K1 phosphorylates MAPK3 at Thr-202 and Tyr-204'
trips_processor = trips.process_text(sentence)
```

The *trips_processor* object has a *statements* attribute which contains a list of INDRA Statements extracted from the sentence.

3.2.2 Reading a PubMed Central article with REACH

In this example, a full paper from PubMed Central is processed. The paper's PMC ID is PMC8511698.

```
from indra.sources import reach
reach_processor = reach.process_pmc('PMC8511698')
```

The *reach_processor* object has a *statements* attribute which contains a list of INDRA Statements extracted from the paper.

3.2.3 Getting the neighborhood of proteins from the BEL Large Corpus

In this example, we search the neighborhood of the KRAS and BRAF proteins in the BEL Large Corpus.

```
from indra.sources import bel
bel_processor = bel.process_pybel_neighborhood(['KRAS', 'BRAF'])
```

The *bel_processor* object has a *statements* attribute which contains a list of INDRA Statements extracted from the queried neighborhood.

3.2.4 Constructing INDRA Statements manually

It is possible to construct INDRA Statements manually or in scripts. The following is a basic example in which we instantiate a Phosphorylation Statement between BRAF and MAP2K1.

```
from indra.statements import Phosphorylation, Agent
braf = Agent('BRAF')
map2k1 = Agent('MAP2K1')
stmt = Phosphorylation(braf, map2k1)
```

3.2.5 Assembling a PySB model and exporting to SBML

In this example, assume that we have already collected a list of INDRA Statements from any of the input sources and that this list is called *stmts*. We will instantiate a PysbAssembler, which produces a PySB model from INDRA Statements.

```
from indra.assemblers.pysb import PysbAssembler
pa = PysbAssembler()
pa.add_statements(stmts)
model = pa.make_model()
```

Here the *model* variable is a PySB Model object representing a rule-based executable model, which can be further manipulated, simulated, saved and exported to other formats.

For instance, exporting the model to SBML format can be done as

```
sbml_model = pa.export_model('sbml')
```

which gives an SBML model string in the *sbml_model* variable, or as

```
pa.export_model('sbml', file_name='model.sbml')
```

which writes the SBML model into the *model.sbml* file. Other formats for export that are supported include BNGL, Kappa and Matlab. For a full list, see the PySB export module.

3.2.6 Exporting Statements as an IndraNet Graph

In this example we again assume that there already exists a variable called *stmts*, containing a list of statements. We will import the *IndraNetAssembler* that produces an IndraNet object, which is a networkx MultiDiGraph representations of the statements, each edge representing a statement and each node being an agent.

```
from indra.assemblers.indranet import IndraNetAssembler
indranet_assembler = IndraNetAssembler(statements=stmts)
indranet = indranet_assembler.make_model()
```

The *indranet* object is an instance of a child class of a networkx graph object, making all networkx graph methods available for the indranet object. Each edge in the has an edge dictionary with meta data from the statement.

The *indranet* graph has methods to map it to other graph types. Here we export it to a signed graph which is represents directed edges with positive or negative polarity signs:

```
signed_graph = indranet.to_signed_graph()
```

Read more about the *IndraNetAssembler* in the documentation.

3.3 See More

For a longer example of using INDRA in an end-to-end pipeline, from getting content from different sources to assembling different output models, see the tutorial "Assembling everything known about a particular gene".

More tutorials are available in the tutorials section.

CHAPTER

FOUR

INDRA MODULES REFERENCE

4.1 INDRA Statements (`indra.statements`)

4.1.1 General information and statement types

Statements represent mechanistic relationships between biological agents.

Statement classes follow an inheritance hierarchy, with all Statement types inheriting from the parent class *Statement*. At the next level in the hierarchy are the following classes:

Open Domain

- *Event*
- *Influence*
- *Association*

Biological Domain

- *Complex*
- *Modification*
- *SelfModification*
- *RegulateActivity*
- *RegulateAmount*
- *ActiveForm*
- *Translocation*
- *Gef*
- *Gap*
- *Conversion*

There are several types of Statements representing post-translational modifications that further inherit from *Modification*:

- *Phosphorylation*
- *Dephosphorylation*
- *Ubiquitination*
- *Deubiquitination*
- *Sumoylation*

- *Desumoylation*
- *Hydroxylation*
- *Dehydroxylation*
- *Acetylation*
- *Deacetylation*
- *Glycosylation*
- *Deglycosylation*
- *Farnesylation*
- *Defarnesylation*
- *Geranylgeranylation*
- *Degeranylgeranylation*
- *Palmitoylation*
- *Depalmitoylation*
- *Myristoylation*
- *Demyristoylation*
- *Ribosylation*
- *Deribosylation*
- *Methylation*
- *Demethylation*

There are additional subtypes of *SelfModification*:

- *Autophosphorylation*
- *Transphosphorylation*

Interactions between proteins are often described simply in terms of their effect on a protein's "activity", e.g., "Active MEK activates ERK", or "DUSP6 inactives ERK". These types of relationships are indicated by the *RegulateActivity* abstract base class which has subtypes

- *Activation*
- *Inhibition*

while the *RegulateAmount* abstract base class has subtypes

- *IncreaseAmount*
- *DecreaseAmount*

Statements involve one or more *Concepts*, which, depending on the semantics of the Statement, are typically biological *Agents*, such as proteins, represented by the class *Agent*. (However, :py:class`Influence` statements involve two or more :py:class`Event` objects, each of which takes a :py:class`Concept` as an argument.)

Agents can have several types of context specified on them including

- a specific post-translational modification state (indicated by one or more instances of *ModCondition*),
- other bound Agents (*BoundCondition*),
- mutations (*MutCondition*),

- an activity state (*ActivityCondition*), and
- cellular location

The *active* form of an agent (in terms of its post-translational modifications or bound state) is indicated by an instance of the class *ActiveForm*.

Grounding and DB references

Agents also carry grounding information which links them to database entries. These database references are represented as a dictionary in the *db_refs* attribute of each Agent. The dictionary can have multiple entries. For instance, INDRA's input Processors produce genes and proteins that carry both UniProt and HGNC IDs in db_refs, whenever possible. FamPlex provides a name space for protein families that are typically used in the literature. More information about FamPlex can be found here: https://github.com/sorgerlab/famplex

In general, the capitalized version of any identifiers.org name space (see https://registry.identifiers.org/ for full list) can be used in db_refs with a few cases where INDRA's internal db_refs name space is different from the identifiers.org name space (e.g., UP vs uniprot). These special cases can be programmatically mapped between INDRA and identifiers.org using the *identifiers_mappings* and *identifiers_reverse* dictionaries in the *indra.databases.identifiers* module.

Examples of the most commonly encountered db_refs name spaces and IDs are listed below.

Type	Database	Example
Gene/Protein	HGNC	{'HGNC': '11998'}
Gene/Protein	UniProt	{'UP': 'P04637'}
Protein chain	UniProt	{'UPPRO': 'PRO_0000435839'}
Gene/Protein	Entrez	{'EGID': '5583'}
Gene/Protein family	FamPlex	{'FPLX': 'ERK'}
Gene/Protein family	InterPro	{'IP': 'IPR000308'}
Gene/Protein family	Pfam	{'PF': 'PF00071'}
Gene/Protein family	NextProt family	{'NXPFA': '03114'}
Chemical	ChEBI	{'CHEBI': 'CHEBI:63637'}
Chemical	PubChem	{'PUBCHEM': '42611257'}
Chemical	LINCS	{'LINCS': '42611257'}
Metabolite	HMDB	{'HMDB': 'HMDB00122'}
Process, location, etc.	GO	{'GO': 'GO:0006915'}
Process, disease, etc.	MeSH	{'MESH': 'D008113'}
Disease	Disease Ontology	{'DOID': 'DOID:8659'}
Phenotypic abnormality	Human Pheno. Ont.	{'HP': 'HP:0031296'}
Experimental factors	Exp. Factor Ont.	{'EFO': '0007820'}
General terms	NCIT	{'NCIT': 'C28597'}
Raw text	TEXT	{'TEXT': 'Nf-kappaB'}

The evidence for a given Statement, which could include relevant citations, database identifiers, and passages of text from the scientific literature, is contained in one or more *Evidence* objects associated with the Statement.

JSON serialization of INDRA Statements

Statements can be serialized into JSON and deserialized from JSON to allow their exchange in a platform-independent way. We also provide a JSON schema (see http://json-schema.org to learn about schemas) in https://raw.githubusercontent.com/sorgerlab/indra/master/indra/resources/statements_schema.json which can be used to validate INDRA Statements JSONs.

Some validation tools include:

- **jsonschema** a Python package to validate JSON content with respect to a schema
- **ajv-cli** Available at https://www.npmjs.com/package/ajv-cli Install with "npm install -g ajv-cli" and then validate with: ajv -s statements_schema.json -d file_to_validate.json. This tool provides more sophisticated and better interpretable output than jsonschema.
- **Web based tools** There are a variety of web-based tools for validation with JSON schemas, including https://www.jsonschemavalidator.net

class indra.statements.statements.**Acetylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.AddModification*

Acetylation modification.

class indra.statements.statements.**Activation**(*subj*, *obj*, *obj_activity='activity'*, *evidence=None*)

Bases: *indra.statements.statements.RegulateActivity*

Indicates that a protein activates another protein.

This statement is intended to be used for physical interactions where the mechanism of activation is not explicitly specified, which is often the case for descriptions of mechanisms extracted from the literature.

Parameters

- **subj** (*Agent*) – The agent responsible for the change in activity, i.e., the "upstream" node.
- **obj** (*Agent*) – The agent whose activity is influenced by the subject, i.e., the "downstream" node.
- **obj_activity** (*Optional[str]*) – The activity of the obj Agent that is affected, e.g., its "kinase" activity.
- **evidence** (None or *Evidence* or list of *Evidence*) – Evidence objects in support of the modification.

Examples

MEK (MAP2K1) activates the kinase activity of ERK (MAPK1):

```
>>> mek = Agent('MAP2K1')
>>> erk = Agent('MAPK1')
>>> act = Activation(mek, erk, 'kinase')
```

class indra.statements.statements.**ActiveForm**(*agent*, *activity*, *is_active*, *evidence=None*)

Bases: *indra.statements.statements.Statement*

Specifies conditions causing an Agent to be active or inactive.

Types of conditions influencing a specific type of biochemical activity can include modifications, bound Agents, and mutations.

Parameters

- **agent** (*Agent*) – The Agent in a particular active or inactive state. The sets of ModConditions, BoundConditions, and MutConditions on the given Agent instance indicate the relevant conditions.
- **activity** (*str*) – The type of activity influenced by the given set of conditions, e.g., "kinase".
- **is_active** (*bool*) – Whether the conditions are activating (True) or inactivating (False).

to_json(*use_sbo=False*, *matches_fun=None*)
Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**ActivityCondition**(*activity_type*, *is_active*)
Bases: object

An active or inactive state of a protein.

Examples

Kinase-active MAP2K1:

```
>>> mek_active = Agent('MAP2K1',
...                    activity=ActivityCondition('kinase', True))
```

Transcriptionally inactive FOXO3:

```
>>> foxo_inactive = Agent('FOXO3',
...                       activity=ActivityCondition('transcription', False))
```

Parameters

- **activity_type** (*str*) – The type of activity, e.g. 'kinase'. The basic, unspecified molecular activity is represented as 'activity'. Examples of other activity types are 'kinase', 'phosphatase', 'catalytic', 'transcription', etc.
- **is_active** (*bool*) – Specifies whether the given activity type is present or absent.

class indra.statements.statements.**AddModification**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)
Bases: *indra.statements.statements.Modification*

class indra.statements.statements.**Agent**(*name*, *mods=None*, *activity=None*, *bound_conditions=None*, *mutations=None*, *location=None*, *db_refs=None*)
Bases: *indra.statements.concept.Concept*

A molecular entity, e.g., a protein.

Parameters

- **name** (*str*) – The name of the agent, preferably a canonicalized name such as an HGNC gene name.
- **mods** (list of *ModCondition*) – Modification state of the agent.
- **bound_conditions** (list of *BoundCondition*) – Other agents bound to the agent in this context.
- **mutations** (list of *MutCondition*) – Amino acid mutations of the agent.
- **activity** (*ActivityCondition*) – Activity of the agent.
- **location** (*str*) – Cellular location of the agent. Must be a valid name (e.g. "nucleus") or identifier (e.g. "GO:0005634")for a GO cellular compartment.
- **db_refs** (*dict*) – Dictionary of database identifiers associated with this agent.

entity_matches_key()
Return a key to identify the identity of the Agent not its state.

The key is based on the preferred grounding for the Agent, or if not available, the name of the Agent is used.

Returns The key used to identify the Agent.

Return type str

get_grounding(*ns_order=None*)
Return a tuple of a preferred grounding namespace and ID.

Returns A tuple whose first element is a grounding namespace (HGNC, CHEBI, etc.) and the second element is an identifier in the namespace. If no preferred grounding is available, a tuple of Nones is returned.

Return type tuple

matches_key()
Return a key to identify the identity and state of the Agent.

state_matches_key()
Return a key to identify the state of the Agent.

class indra.statements.statements.**Association**(*members*, *evidence=None*)
Bases: *indra.statements.statements.Complex*

A set of events associated with each other without causal relationship.

Parameters

- **members** (*list of :py:class:Event*) – A list of events associated with each other.
- **evidence** (None or *Evidence* or list of *Evidence*) – Evidence objects in support of the modification.

agent_list(*deep_sorted=False*)
Get the canonicalized agent list.

flip_polarity(*agent_idx*)
If applicable, flip the polarity of the statement

to_json(*use_sbo=False*, *matches_fun=None*)
Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**Autophosphorylation**(*enz*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.SelfModification*

Intramolecular autophosphorylation, i.e., in *cis*.

Examples

p38 bound to TAB1 cis-autophosphorylates itself (see PMID:19155529).

```
>>> tab1 = Agent('TAB1')
>>> p38_tab1 = Agent('P38', bound_conditions=[BoundCondition(tab1)])
>>> autophos = Autophosphorylation(p38_tab1)
```

class indra.statements.statements.**BioContext**(*location=None*, *cell_line=None*, *cell_type=None*, *organ=None*, *disease=None*, *species=None*)

Bases: *indra.statements.context.Context*

An object representing the context of a Statement in biology.

Parameters

- **location** (*Optional[RefContext]*) – Cellular location, typically a sub-cellular compartment.
- **cell_line** (*Optional[RefContext]*) – Cell line context, e.g., a specific cell line, like BT20.
- **cell_type** (*Optional[RefContext]*) – Cell type context, broader than a cell line, like macrophage.
- **organ** (*Optional[RefContext]*) – Organ context.
- **disease** (*Optional[RefContext]*) – Disease context.
- **species** (*Optional[RefContext]*) – Species context.

class indra.statements.statements.**BoundCondition**(*agent*, *is_bound=True*)

Bases: object

Identify Agents bound (or not bound) to a given Agent in a given context.

Parameters

- **agent** (*Agent*) – Instance of Agent.
- **is_bound** (*bool*) – Specifies whether the given Agent is bound or unbound in the current context. Default is True.

Examples

EGFR bound to EGF:

```
>>> egf = Agent('EGF')
>>> egfr = Agent('EGFR', bound_conditions=[BoundCondition(egf)])
```

BRAF *not* bound to a 14-3-3 protein (YWHAB):

```
>>> ywhab = Agent('YWHAB')
>>> braf = Agent('BRAF', bound_conditions=[BoundCondition(ywhab, False)])
```

class indra.statements.statements.**Complex**(*members*, *evidence=None*)

Bases: *indra.statements.statements.Statement*

A set of proteins observed to be in a complex.

Parameters **members** (list of *Agent*) – The set of proteins in the complex.

Examples

BRAF is observed to be in a complex with RAF1:

```
>>> braf = Agent('BRAF')
>>> raf1 = Agent('RAF1')
>>> cplx = Complex([braf, raf1])
```

to_json(*use_sbo=False*, *matches_fun=None*)

Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**Concept**(*name*, *db_refs=None*)

Bases: object

A concept/entity of interest that is the argument of a Statement

Parameters

- **name** (*str*) – The name of the concept, possibly a canonicalized name.
- **db_refs** (*dict*) – Dictionary of database identifiers associated with this concept.

class indra.statements.statements.**Context**

Bases: object

An abstract class for Contexts.

class indra.statements.statements.**Conversion**(*subj*, *obj_from=None*, *obj_to=None*, *evidence=None*)
Bases: *indra.statements.statements.Statement*

Conversion of molecular species mediated by a controller protein.

Parameters

- **subj** (indra.statement.Agent) – The protein mediating the conversion.
- **obj_from** (list of indra.statement.Agent) – The list of molecular species being consumed by the conversion.
- **obj_to** (list of indra.statement.Agent) – The list of molecular species being created by the conversion.
- **evidence** (None or *Evidence* or list of *Evidence*) – Evidence objects in support of the synthesis statement.

to_json(*use_sbo=False*, *matches_fun=None*)
Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**Deacetylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)
Bases: *indra.statements.statements.RemoveModification*

Deacetylation modification.

class indra.statements.statements.**DecreaseAmount**(*subj*, *obj*, *evidence=None*)
Bases: *indra.statements.statements.RegulateAmount*

Degradation of a protein, possibly mediated by another protein.

Note that this statement can also be used to represent inhibitors of synthesis (e.g., cycloheximide).

Parameters

- **subj** (indra.statement.Agent) – The protein mediating the degradation.
- **obj** (indra.statement.Agent) – The protein that is degraded.
- **evidence** (None or *Evidence* or list of *Evidence*) – Evidence objects in support of the degradation statement.

class indra.statements.statements.**Defarnesylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)
Bases: *indra.statements.statements.RemoveModification*

Defarnesylation modification.

class indra.statements.statements.**Degeranylgeranylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)
Bases: *indra.statements.statements.RemoveModification*

Degeranylgeranylation modification.

class indra.statements.statements.**Deglycosylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.RemoveModification*

Deglycosylation modification.

class indra.statements.statements.**Dehydroxylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.RemoveModification*

Dehydroxylation modification.

class indra.statements.statements.**Demethylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.RemoveModification*

Demethylation modification.

class indra.statements.statements.**Demyristoylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.RemoveModification*

Demyristoylation modification.

class indra.statements.statements.**Depalmitoylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.RemoveModification*

Depalmitoylation modification.

class indra.statements.statements.**Dephosphorylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.RemoveModification*

Dephosphorylation modification.

Examples

DUSP6 dephosphorylates ERK (MAPK1) at T185:

```
>>> dusp6 = Agent('DUSP6')
>>> erk = Agent('MAPK1')
>>> dephos = Dephosphorylation(dusp6, erk, 'T', '185')
```

class indra.statements.statements.**Deribosylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.RemoveModification*

Deribosylation modification.

class indra.statements.statements.**Desumoylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.RemoveModification*

Desumoylation modification.

class indra.statements.statements.**Deubiquitination**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.RemoveModification*

Deubiquitination modification.

class indra.statements.statements.**Event**(*concept*, *delta=None*, *context=None*, *evidence=None*, *supports=None*, *supported_by=None*)

Bases: *indra.statements.statements.Statement*

An event representing the change of a Concept.

concept
The concept over which the event is defined.

Type *indra.statements.concept.Concept*

delta
Represents a change in the concept, with a polarity and an adjectives entry.

Type indra.statements.delta.Delta

context
The context associated with the event.

Type *indra.statements.context.Context*

flip_polarity(*agent_idx=None*)
If applicable, flip the polarity of the statement

to_json(*with_evidence=True*, *use_sbo=False*, *matches_fun=None*)
Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**Evidence**(*source_api=None*, *source_id=None*, *pmid=None*, *text=None*, *annotations=None*, *epistemics=None*, *context=None*, *text_refs=None*)

Bases: object

Container for evidence supporting a given statement.

Parameters

- **source_api** (*str or None*) – String identifying the INDRA API used to capture the statement, e.g., 'trips', 'biopax', 'bel'.
- **source_id** (*str or None*) – For statements drawn from databases, ID of the database entity corresponding to the statement.
- **pmid** (*str or None*) – String indicating the Pubmed ID of the source of the statement.
- **text** (*str*) – Natural language text supporting the statement.
- **annotations** (*dict*) – Dictionary containing additional information on the context of the statement, e.g., species, cell line, tissue type, etc. The entries may vary depending on the source of the information.

- **epistemics** (*dict*) – A dictionary describing various forms of epistemic certainty associated with the statement.
- **context** (*Context or None*) – A context object
- **text_refs** (*dict*) – A dictionary of various reference ids to the source text, e.g. DOI, PMID, URL, etc.

There are some attributes which are not set by the parameters above:

source_hash [int] A hash calculated from the evidence text, source api, and pmid and/or source_id if available. This is generated automatcially when the object is instantiated.

stmt_tag [int] This is a hash calculated by a Statement to which this evidence refers, and is set by said Statement. It is useful for tracing ownership of an Evidence object.

get_source_hash(*refresh=False*)
Get a hash based off of the source of this statement.

The resulting value is stored in the source_hash attribute of the class and is preserved in the json dictionary.

to_json()
Convert the evidence object into a JSON dict.

class indra.statements.statements.**Farnesylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)
Bases: *indra.statements.statements.AddModification*

Farnesylation modification.

class indra.statements.statements.**Gap**(*gap*, *ras*, *evidence=None*)
Bases: *indra.statements.statements.Statement*

Acceleration of a GTPase protein's GTP hydrolysis rate by a GAP.

Represents the generic process by which a GTPase activating protein (GAP) catalyzes GTP hydrolysis by a particular small GTPase protein.

Parameters

- **gap** (*Agent*) – The GTPase activating protein.
- **ras** (*Agent*) – The GTPase protein.

Examples

RASA1 catalyzes GTP hydrolysis on KRAS:

```
>>> rasa1 = Agent('RASA1')
>>> kras = Agent('KRAS')
>>> gap = Gap(rasa1, kras)
```

to_json(*use_sbo=False*, *matches_fun=None*)
Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**Gef**(*gef*, *ras*, *evidence=None*)
Bases: *indra.statements.statements.Statement*

Exchange of GTP for GDP on a small GTPase protein mediated by a GEF.

Represents the generic process by which a guanosine exchange factor (GEF) catalyzes nucleotide exchange on a GTPase protein.

Parameters

- **gef** (*Agent*) – The guanosine exchange factor.
- **ras** (*Agent*) – The GTPase protein.

Examples

SOS1 catalyzes nucleotide exchange on KRAS:

```
>>> sos = Agent('SOS1')
>>> kras = Agent('KRAS')
>>> gef = Gef(sos, kras)
```

to_json(*use_sbo=False*, *matches_fun=None*)
Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**Geranylgeranylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)
Bases: *indra.statements.statements.AddModification*

Geranylgeranylation modification.

class indra.statements.statements.**Glycosylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)
Bases: *indra.statements.statements.AddModification*

Glycosylation modification.

class indra.statements.statements.**GtpActivation**(*subj*, *obj*, *obj_activity='activity'*, *evidence=None*)
Bases: *indra.statements.statements.Activation*

class indra.statements.statements.**HasActivity**(*agent*, *activity*, *has_activity*, *evidence=None*)
Bases: *indra.statements.statements.Statement*

States that an Agent has or doesn't have a given activity type.

With this Statement, one cane express that a given protein is a kinase, or, for instance, that it is a transcription factor. It is also possible to construct negative statements with which one epxresses, for instance, that a given protein is not a kinase.

Parameters

- **agent** (*Agent*) – The Agent that that statement is about. Note that the detailed state of the Agent is not relevant for this type of statement.
- **activity** (*str*) – The type of activity, e.g., "kinase".
- **has_activity** (*bool*) – Whether the given Agent has the given activity (True) or not (False).

class indra.statements.statements.**Hydroxylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.AddModification*

Hydroxylation modification.

class indra.statements.statements.**IncreaseAmount**(*subj*, *obj*, *evidence=None*)

Bases: *indra.statements.statements.RegulateAmount*

Synthesis of a protein, possibly mediated by another protein.

Parameters

- **subj** (indra.statement.Agent) – The protein mediating the synthesis.
- **obj** (indra.statement.Agent) – The protein that is synthesized.
- **evidence** (None or *Evidence* or list of *Evidence*) – Evidence objects in support of the synthesis statement.

class indra.statements.statements.**Influence**(*subj*, *obj*, *evidence=None*)

Bases: *indra.statements.statements.Statement*

An influence on the quantity of a concept of interest.

Parameters

- **subj** (indra.statement.Event) – The event which acts as the influencer.
- **obj** (indra.statement.Event) – The event which acts as the influencee.
- **evidence** (None or *Evidence* or list of *Evidence*) – Evidence objects in support of the statement.

agent_list(*deep_sorted=False*)

Get the canonicalized agent list.

flip_polarity(*agent_idx*)

If applicable, flip the polarity of the statement

to_json(*use_sbo=False*, *matches_fun=None*)

Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**Inhibition**(*subj*, *obj*, *obj_activity='activity'*, *evidence=None*)
Bases: *indra.statements.statements.RegulateActivity*

Indicates that a protein inhibits or deactivates another protein.

This statement is intended to be used for physical interactions where the mechanism of inhibition is not explicitly specified, which is often the case for descriptions of mechanisms extracted from the literature.

Parameters

- **subj** (*Agent*) – The agent responsible for the change in activity, i.e., the "upstream" node.
- **obj** (*Agent*) – The agent whose activity is influenced by the subject, i.e., the "downstream" node.
- **obj_activity** (*Optional[str]*) – The activity of the obj Agent that is affected, e.g., its "kinase" activity.
- **evidence** (None or *Evidence* or list of *Evidence*) – Evidence objects in support of the modification.

exception indra.statements.statements.**InputError**
Bases: Exception

exception indra.statements.statements.**InvalidLocationError**(*name*)
Bases: ValueError

Invalid cellular component name.

exception indra.statements.statements.**InvalidResidueError**(*name*)
Bases: ValueError

Invalid residue (amino acid) name.

class indra.statements.statements.**Methylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)
Bases: *indra.statements.statements.AddModification*

Methylation modification.

class indra.statements.statements.**Migration**(*concept*, *delta=None*, *context=None*, *evidence=None*, *supports=None*, *supported_by=None*)
Bases: *indra.statements.statements.Event*

A special class of Event representing Migration.

class indra.statements.statements.**ModCondition**(*mod_type*, *residue=None*, *position=None*, *is_modified=True*)
Bases: object

Post-translational modification state at an amino acid position.

Parameters

- **mod_type** (*str*) – The type of post-translational modification, e.g., 'phosphorylation'. Valid modification types currently include: 'phosphorylation', 'ubiquitination', 'sumoylation', 'hydroxylation', and 'acetylation'. If an invalid modification type is passed an InvalidModTypeError is raised.
- **residue** (*str or None*) – String indicating the modified amino acid, e.g., 'Y' or 'tyrosine'. If None, indicates that the residue at the modification site is unknown or unspecified.

- **position** (*str or None*) – String indicating the position of the modified amino acid, e.g., '202'. If None, indicates that the position is unknown or unspecified.
- **is_modified** (*bool*) – Specifies whether the modification is present or absent. Setting the flag specifies that the Agent with the ModCondition is unmodified at the site.

Examples

Doubly-phosphorylated MEK (MAP2K1):

```
>>> phospho_mek = Agent('MAP2K1', mods=[
... ModCondition('phosphorylation', 'S', '202'),
... ModCondition('phosphorylation', 'S', '204')])
```

ERK (MAPK1) unphosphorylated at tyrosine 187:

```
>>> unphos_erk = Agent('MAPK1', mods=(
... ModCondition('phosphorylation', 'Y', '187', is_modified=False)))
```

class indra.statements.statements.**Modification**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.Statement*

Generic statement representing the modification of a protein.

Parameters

- **enz** (indra.statement.Agent) – The enzyme involved in the modification.
- **sub** (indra.statement.Agent) – The substrate of the modification.
- **residue** (*str or None*) – The amino acid residue being modified, or None if it is unknown or unspecified.
- **position** (*str or None*) – The position of the modified amino acid, or None if it is unknown or unspecified.
- **evidence** (None or *Evidence* or list of *Evidence*) – Evidence objects in support of the modification.

to_json(*use_sbo=False*, *matches_fun=None*)

Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**MovementContext**(*locations=None*, *time=None*)

Bases: *indra.statements.context.Context*

An object representing the context of a movement between start and end points in time.

Parameters

- **locations** (*Optional[list[dict]]*) – A list of dictionaries each containing a RefContext object representing geographical location context and its role (e.g. 'origin', 'destination', etc.)
- **time** (*Optional[TimeContext]*) – A TimeContext object representing the temporal context of the Statement.

class indra.statements.statements.**MutCondition**(*position*, *residue_from*, *residue_to=None*)

Bases: object

Mutation state of an amino acid position of an Agent.

Parameters

- **position** (*str*) – Residue position of the mutation in the protein sequence.
- **residue_from** (*str*) – Wild-type (unmodified) amino acid residue at the given position.
- **residue_to** (*str*) – Amino acid at the position resulting from the mutation.

Examples

Represent EGFR with a L858R mutation:

```
>>> egfr_mutant = Agent('EGFR', mutations=[MutCondition('858', 'L', 'R')])
```

class indra.statements.statements.**Myristoylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.AddModification*

Myristoylation modification.

exception indra.statements.statements.**NotAStatementName**

Bases: Exception

class indra.statements.statements.**Palmitoylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.AddModification*

Palmitoylation modification.

class indra.statements.statements.**Phosphorylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.AddModification*

Phosphorylation modification.

Examples

MEK (MAP2K1) phosphorylates ERK (MAPK1) at threonine 185:

```
>>> mek = Agent('MAP2K1')
>>> erk = Agent('MAPK1')
>>> phos = Phosphorylation(mek, erk, 'T', '185')
```

class indra.statements.statements.**QualitativeDelta**(*polarity=None*, *adjectives=None*)

Bases: indra.statements.delta.Delta

Qualitative delta defining an Event.

Parameters

- **polarity** (*1, -1 or None*) – Polarity of an Event.
- **adjectives** (*list[str]*) – Adjectives describing an Event.

class indra.statements.statements.**QuantitativeState**(*entity=None*, *value=None*, *unit=None*, *modifier=None*, *text=None*, *polarity=None*)

Bases: indra.statements.delta.Delta

An object representing numerical value of something.

Parameters

- **entity** (*str*) – An entity to capture the quantity of.
- **value** (*float or int*) – Quantity of a unit (or range?)
- **unit** (*str*) – Measurement unit of value (e.g. absolute, daily, percentage, etc.)
- **modifier** (*str*) – Modifier to value (e.g. more than, at least, approximately, etc.)
- **text** (*str*) – Natural language text describing quantitative state.
- **polarity** (*1, -1 or None*) – Polarity of an Event.

static convert_unit(*source_unit*, *target_unit*, *source_value*, *source_period=None*, *target_period=None*)
Convert value per unit from source to target unit. If a unit is absolute, total timedelta period has to be provided. If a unit is a month or a year, it is recommended to pass timedelta period object directly, if not provided, the approximation will be used.

static from_seconds(*value_per_second*, *period*)
Get total value per given period given timedelta period object and value per second.

static value_per_second(*value*, *period*)
Get value per second given total value per period and a timedelta period object.

class indra.statements.statements.**RefContext**(*name=None*, *db_refs=None*)

Bases: object

An object representing a context with a name and references.

Parameters

- **name** (*Optional[str]*) – The name of the given context. In some cases a text name will not be available so this is an optional parameter with the default being None.
- **db_refs** (*Optional[dict]*) – A dictionary where each key is a namespace and each value is an identifier in that namespace, similar to the db_refs associated with Concepts/Agents.

class indra.statements.statements.**RegulateActivity**

Bases: *indra.statements.statements.Statement*

Regulation of activity.

This class implements shared functionality of Activation and Inhibition statements and it should not be instantiated directly.

to_json(*use_sbo=False*, *matches_fun=None*)
Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False

- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**RegulateAmount**(*subj*, *obj*, *evidence=None*)

Bases: *indra.statements.statements.Statement*

Superclass handling operations on directed, two-element interactions.

to_json(*use_sbo=False*, *matches_fun=None*)

Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**RemoveModification**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.Modification*

class indra.statements.statements.**Ribosylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.AddModification*

Ribosylation modification.

class indra.statements.statements.**SelfModification**(*enz*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.statements.Statement*

Generic statement representing the self-modification of a protein.

Parameters

- **enz** (indra.statement.Agent) – The enzyme involved in the modification, which is also the substrate.
- **residue** (*str or None*) – The amino acid residue being modified, or None if it is unknown or unspecified.
- **position** (*str or None*) – The position of the modified amino acid, or None if it is unknown or unspecified.
- **evidence** (None or *Evidence* or list of *Evidence*) – Evidence objects in support of the modification.

to_json(*use_sbo=False*, *matches_fun=None*)

Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**Statement**(*evidence=None*, *supports=None*, *supported_by=None*)

Bases: object

The parent class of all statements.

Parameters

- **evidence** (None or *Evidence* or list of *Evidence*) – If a list of Evidence objects is passed to the constructor, the value is set to this list. If a bare Evidence object is passed, it is enclosed in a list. If no evidence is passed (the default), the value is set to an empty list.
- **supports** (list of *Statement*) – Statements that this Statement supports.
- **supported_by** (list of *Statement*) – Statements supported by this statement.

agent_list(*deep_sorted=False*)

Get the canonicalized agent list.

flip_polarity(*agent_idx=None*)

If applicable, flip the polarity of the statement

get_hash(*shallow=True*, *refresh=False*, *matches_fun=None*)

Get a hash for this Statement.

There are two types of hash, "shallow" and "full". A shallow hash is as unique as the information carried by the statement, i.e. it is a hash of the *matches_key*. This means that differences in source, evidence, and so on are not included. As such, it is a shorter hash (14 nibbles). The odds of a collision among all the statements we expect to encounter (well under 10^8) is ~10^-9 (1 in a billion). Checks for collisions can be done by using the matches keys.

A full hash includes, in addition to the matches key, information from the evidence of the statement. These hashes will be equal if the two Statements came from the same sentences, extracted by the same reader, from the same source. These hashes are correspondingly longer (16 nibbles). The odds of a collision for an expected less than 10^10 extractions is ~10^-9 (1 in a billion).

Note that a hash of the Python object will also include the *uuid*, so it will always be unique for every object.

Parameters

- **shallow** (*bool*) – Choose between the shallow and full hashes described above. Default is true (e.g. a shallow hash).
- **refresh** (*bool*) – Used to get a new copy of the hash. Default is false, so the hash, if it has been already created, will be read from the attribute. This is primarily used for speed testing.
- **matches_fun** (*Optional[function]*) – A function which takes a Statement as argument and returns a string matches key which is then hashed. If not provided the Statement's built-in matches_key method is used.

Returns **hash** – A long integer hash.

Return type int

make_generic_copy(*deeply=False*)
Make a new matching Statement with no provenance.

All agents and other attributes besides evidence, uuid, supports, and supported_by will be copied over, and a new uuid will be assigned. Thus, the new Statement will satisfy *new_stmt.matches(old_stmt)*.

If *deeply* is set to True, all the attributes will be deep-copied, which is comparatively slow. Otherwise, attributes of this statement may be altered by changes to the new matching statement.

real_agent_list()
Return all agents in the statement that are not None.

to_graph()
Return Statement as a networkx graph.

to_json(*use_sbo=False*, *matches_fun=None*)
Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**Sumoylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)
Bases: *indra.statements.statements.AddModification*

Sumoylation modification.

class indra.statements.statements.**TimeContext**(*text=None*, *start=None*, *end=None*, *duration=None*)
Bases: object

An object representing the time context of a Statement

Parameters

- **text** (*Optional[str]*) – A string representation of the time constraint, typically as seen in text.
- **start** (*Optional[datetime]*) – A *datetime* object representing the start time
- **end** (*Optional[datetime]*) – A *datetime* object representing the end time
- **duration** (*int*) – The duration of the time constraint in seconds

class indra.statements.statements.**Translocation**(*agent*, *from_location=None*, *to_location=None*, *evidence=None*)
Bases: *indra.statements.statements.Statement*

The translocation of a molecular agent from one location to another.

Parameters

- **agent** (*Agent*) – The agent which translocates.
- **from_location** (*Optional[str]*) – The location from which the agent translocates. This must be a valid GO cellular component name (e.g. "cytoplasm") or ID (e.g. "GO:0005737").

- **to_location** (*Optional[str]*) – The location to which the agent translocates. This must be a valid GO cellular component name or ID.

to_json(*use_sbo=False*, *matches_fun=None*)
Return serialized Statement as a JSON dict.

Parameters

- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – The JSON-serialized INDRA Statement.

Return type dict

class indra.statements.statements.**Transphosphorylation**(*enz*, *residue=None*, *position=None*, *evidence=None*)
Bases: *indra.statements.statements.SelfModification*

Autophosphorylation in *trans*.

Transphosphorylation assumes that a kinase is already bound to a substrate (usually of the same molecular species), and phosphorylates it in an intra-molecular fashion. The enz property of the statement must have exactly one bound_conditions entry, and we assume that enz phosphorylates this molecule. The bound_neg property is ignored here.

class indra.statements.statements.**Ubiquitination**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)
Bases: *indra.statements.statements.AddModification*

Ubiquitination modification.

class indra.statements.statements.**Unresolved**(*uuid_str=None*, *shallow_hash=None*, *full_hash=None*)
Bases: *indra.statements.statements.Statement*

A special statement type used in support when a uuid can't be resolved.

When using the *stmts_from_json* method, it is sometimes not possible to resolve the uuid found in *support* and *supported_by* in the json representation of an indra statement. When this happens, this class is used as a placeholder, carrying only the uuid of the statement.

exception indra.statements.statements.**UnresolvedUuidError**
Bases: Exception

class indra.statements.statements.**WorldContext**(*time=None*, *geo_location=None*)
Bases: *indra.statements.context.Context*

An object representing the context of a Statement in time and space.

Parameters

- **time** (*Optional[TimeContext]*) – A TimeContext object representing the temporal context of the Statement.
- **geo_location** (*Optional[RefContext]*) – The geographical location context represented as a RefContext

indra.statements.statements.**draw_stmt_graph**(*stmts*)
Render the attributes of a list of Statements as directed graphs.

The layout works well for a single Statement or a few Statements at a time. This function displays the plot of the graph using plt.show().

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of one or more INDRA Statements whose attribute graph should be drawn.

indra.statements.statements.**get_all_descendants**(*parent*)
Get all the descendants of a parent class, recursively.

indra.statements.statements.**get_statement_by_name**(*stmt_name*)
Get a statement class given the name of the statement class.

indra.statements.statements.**get_unresolved_support_uuids**(*stmts*)
Get uuids unresolved in support from stmts from stmts_from_json.

indra.statements.statements.**get_valid_residue**(*residue*)
Check if the given string represents a valid amino acid residue.

indra.statements.statements.**make_hash**(*s*, *n_bytes*)
Make the hash from a matches key.

indra.statements.statements.**make_statement_camel**(*stmt_name*)
Makes a statement name match the case of the corresponding statement.

indra.statements.statements.**mk_str**(*mk*)
Replace class path for backwards compatibility of matches keys.

indra.statements.statements.**pretty_print_stmts**(*stmt_list*, *stmt_limit=None*, *ev_limit=5*, *width=None*)
Print a formatted list of statements along with evidence text.

Requires the tabulate package (https://pypi.org/project/tabulate).

Parameters

- **stmt_list** (*List[Statement]*) – The list of INDRA Statements to be printed.
- **stmt_limit** (*Optional[int]*) – The maximum number of INDRA Statements to be printed. If None, all Statements are printed. (Default is None)
- **ev_limit** (*Optional[int]*) – The maximum number of Evidence to print for each Statement. If None, all evidence will be printed for each Statement. (Default is 5)
- **width** (*Optional[int]*) – Manually set the width of the table. If *None* the function will try to match the current terminal width using *os.get_terminal_size()*. If this fails the width defaults to 80 characters. The maximum width can be controlled by setting `pretty_print_max_width` using the *set_pretty_print_max_width()* function. This is useful in Jupyter notebooks where the environment returns a terminal size of 80 characters regardless of the width of the window. (Default is None).

Return type None

indra.statements.statements.**print_stmt_summary**(*statements*)
Print a summary of a list of statements by statement type

Requires the tabulate package (https://pypi.org/project/tabulate).

Parameters **statements** (*List[Statement]*) – The list of INDRA Statements to be printed.

indra.statements.statements.**set_pretty_print_max_width**(*new_max*)
Set the max display width for pretty prints, in characters.

indra.statements.statements.**stmt_type**(*obj*, *mk=True*)
Return standardized, backwards compatible object type String.

This is a temporary solution to make sure type comparisons and matches keys of Statements and related classes are backwards compatible.

indra.statements.statements.**stmts_from_json**(*json_in*, *on_missing_support='handle'*)
Get a list of Statements from Statement jsons.

In the case of pre-assembled Statements which have *supports* and *supported_by* lists, the uuids will be replaced with references to Statement objects from the json, where possible. The method of handling missing support is controled by the *on_missing_support* key-word argument.

Parameters

- **json_in** (*iterable[dict]*) – A json list containing json dict representations of INDRA Statements, as produced by the *to_json* methods of subclasses of Statement, or equivalently by *stmts_to_json*.
- **on_missing_support** (*Optional[str]*) – Handles the behavior when a uuid reference in *supports* or *supported_by* attribute cannot be resolved. This happens because uuids can only be linked to Statements contained in the *json_in* list, and some may be missing if only some of all the Statements from pre- assembly are contained in the list.

  Options:

  - *'handle'* : (default) convert unresolved uuids into *Unresolved* Statement objects.
  - *'ignore'* : Simply omit any uuids that cannot be linked to any Statements in the list.
  - *'error'* : Raise an error upon hitting an un-linkable uuid.

Returns **stmts** – A list of INDRA Statements.

Return type list[*Statement*]

indra.statements.statements.**stmts_from_json_file**(*fname*, *format='json'*)
Return a list of statements loaded from a JSON file.

Parameters

- **fname** (Union[str, Path, PathLike]) – Path to the JSON file to load statements from.
- **format** (*Optional[str]*) – One of 'json' to assume regular JSON formatting or 'jsonl' assuming each statement is on a new line.

Returns The list of INDRA Statements loaded from the JSOn file.

Return type list[indra.statements.Statement]

indra.statements.statements.**stmts_to_json**(*stmts_in*, *use_sbo=False*, *matches_fun=None*)
Return the JSON-serialized form of one or more INDRA Statements.

Parameters

- **stmts_in** (*Statement or list[Statement]*) – A Statement or list of Statement objects to serialize into JSON.
- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – JSON-serialized INDRA Statements.

Return type dict

indra.statements.statements.**stmts_to_json_file**(*stmts*, *fname*, *format='json'*, ***kwargs*)
Serialize a list of INDRA Statements into a JSON file.

Parameters

- **stmts** (*list[indra.statement.Statements]*) – The list of INDRA Statements to serialize into the JSON file.
- **fname** (Union[str, Path, PathLike]) – Path to the JSON file to serialize Statements into.
- **format** (*Optional[str]*) – One of 'json' to use regular JSON with indent=1 formatting or 'jsonl' to put each statement on a new line without indents.

4.1.2 Agents (indra.statements.agent)

class indra.statements.agent.**ActivityCondition**(*activity_type*, *is_active*)
Bases: object

An active or inactive state of a protein.

Examples

Kinase-active MAP2K1:

```
>>> mek_active = Agent('MAP2K1',
...                    activity=ActivityCondition('kinase', True))
```

Transcriptionally inactive FOXO3:

```
>>> foxo_inactive = Agent('FOXO3',
...                       activity=ActivityCondition('transcription', False))
```

Parameters

- **activity_type** (*str*) – The type of activity, e.g. 'kinase'. The basic, unspecified molecular activity is represented as 'activity'. Examples of other activity types are 'kinase', 'phosphatase', 'catalytic', 'transcription', etc.
- **is_active** (*bool*) – Specifies whether the given activity type is present or absent.

class indra.statements.agent.**Agent**(*name*, *mods=None*, *activity=None*, *bound_conditions=None*, *mutations=None*, *location=None*, *db_refs=None*)
Bases: *indra.statements.concept.Concept*

A molecular entity, e.g., a protein.

Parameters

- **name** (*str*) – The name of the agent, preferably a canonicalized name such as an HGNC gene name.
- **mods** (list of *ModCondition*) – Modification state of the agent.
- **bound_conditions** (list of *BoundCondition*) – Other agents bound to the agent in this context.
- **mutations** (list of *MutCondition*) – Amino acid mutations of the agent.
- **activity** (*ActivityCondition*) – Activity of the agent.

- **location** (*str*) – Cellular location of the agent. Must be a valid name (e.g. "nucleus") or identifier (e.g. "GO:0005634")for a GO cellular compartment.
- **db_refs** (*dict*) – Dictionary of database identifiers associated with this agent.

entity_matches_key()
Return a key to identify the identity of the Agent not its state.

The key is based on the preferred grounding for the Agent, or if not available, the name of the Agent is used.

Returns The key used to identify the Agent.

Return type str

get_grounding(*ns_order=None*)
Return a tuple of a preferred grounding namespace and ID.

Returns A tuple whose first element is a grounding namespace (HGNC, CHEBI, etc.) and the second element is an identifier in the namespace. If no preferred grounding is available, a tuple of Nones is returned.

Return type tuple

matches_key()
Return a key to identify the identity and state of the Agent.

state_matches_key()
Return a key to identify the state of the Agent.

class indra.statements.agent.**BoundCondition**(*agent*, *is_bound=True*)
Bases: object

Identify Agents bound (or not bound) to a given Agent in a given context.

Parameters

- **agent** (*Agent*) – Instance of Agent.
- **is_bound** (*bool*) – Specifies whether the given Agent is bound or unbound in the current context. Default is True.

Examples

EGFR bound to EGF:

```
>>> egf = Agent('EGF')
>>> egfr = Agent('EGFR', bound_conditions=[BoundCondition(egf)])
```

BRAF *not* bound to a 14-3-3 protein (YWHAB):

```
>>> ywhab = Agent('YWHAB')
>>> braf = Agent('BRAF', bound_conditions=[BoundCondition(ywhab, False)])
```

class indra.statements.agent.**ModCondition**(*mod_type*, *residue=None*, *position=None*, *is_modified=True*)
Bases: object

Post-translational modification state at an amino acid position.

Parameters

- **mod_type** (*str*) – The type of post-translational modification, e.g., 'phosphorylation'. Valid modification types currently include: 'phosphorylation', 'ubiquitination', 'sumoylation', 'hydroxylation', and 'acetylation'. If an invalid modification type is passed an InvalidModTypeError is raised.
- **residue** (*str or None*) – String indicating the modified amino acid, e.g., 'Y' or 'tyrosine'. If None, indicates that the residue at the modification site is unknown or unspecified.
- **position** (*str or None*) – String indicating the position of the modified amino acid, e.g., '202'. If None, indicates that the position is unknown or unspecified.
- **is_modified** (*bool*) – Specifies whether the modification is present or absent. Setting the flag specifies that the Agent with the ModCondition is unmodified at the site.

Examples

Doubly-phosphorylated MEK (MAP2K1):

```
>>> phospho_mek = Agent('MAP2K1', mods=[
... ModCondition('phosphorylation', 'S', '202'),
... ModCondition('phosphorylation', 'S', '204')])
```

ERK (MAPK1) unphosphorylated at tyrosine 187:

```
>>> unphos_erk = Agent('MAPK1', mods=(
... ModCondition('phosphorylation', 'Y', '187', is_modified=False)))
```

class indra.statements.agent.**MutCondition**(*position*, *residue_from*, *residue_to=None*)

Bases: object

Mutation state of an amino acid position of an Agent.

Parameters

- **position** (*str*) – Residue position of the mutation in the protein sequence.
- **residue_from** (*str*) – Wild-type (unmodified) amino acid residue at the given position.
- **residue_to** (*str*) – Amino acid at the position resulting from the mutation.

Examples

Represent EGFR with a L858R mutation:

```
>>> egfr_mutant = Agent('EGFR', mutations=[MutCondition('858', 'L', 'R')])
```

4.1.3 Concepts (indra.statements.concept)

class indra.statements.concept.**Concept**(*name*, *db_refs=None*)

Bases: object

A concept/entity of interest that is the argument of a Statement

Parameters

- **name** (*str*) – The name of the concept, possibly a canonicalized name.
- **db_refs** (*dict*) – Dictionary of database identifiers associated with this concept.

indra.statements.concept.**compositional_sort_key**(*entry*)
Return a sort key from a compositional grounding entry

indra.statements.concept.**get_sorted_compositional_groundings**(*groundings*)
Return the compositional groundings sorted starting from the top.

indra.statements.concept.**get_top_compositional_grounding**(*groundings*)
Return the highest ranking compositional grounding entry.

4.1.4 Evidence (indra.statements.evidence)

class indra.statements.evidence.**Evidence**(*source_api=None*, *source_id=None*, *pmid=None*, *text=None*, *annotations=None*, *epistemics=None*, *context=None*, *text_refs=None*)

Bases: object

Container for evidence supporting a given statement.

Parameters

- **source_api** (*str or None*) – String identifying the INDRA API used to capture the statement, e.g., 'trips', 'biopax', 'bel'.
- **source_id** (*str or None*) – For statements drawn from databases, ID of the database entity corresponding to the statement.
- **pmid** (*str or None*) – String indicating the Pubmed ID of the source of the statement.
- **text** (*str*) – Natural language text supporting the statement.
- **annotations** (*dict*) – Dictionary containing additional information on the context of the statement, e.g., species, cell line, tissue type, etc. The entries may vary depending on the source of the information.
- **epistemics** (*dict*) – A dictionary describing various forms of epistemic certainty associated with the statement.
- **context** (*Context or None*) – A context object
- **text_refs** (*dict*) – A dictionary of various reference ids to the source text, e.g. DOI, PMID, URL, etc.

There are some attributes which are not set by the parameters above:

source_hash [int] A hash calculated from the evidence text, source api, and pmid and/or source_id if available. This is generated automatcially when the object is instantiated.

stmt_tag [int] This is a hash calculated by a Statement to which this evidence refers, and is set by said Statement. It is useful for tracing ownership of an Evidence object.

get_source_hash(*refresh=False*)
Get a hash based off of the source of this statement.

The resulting value is stored in the source_hash attribute of the class and is preserved in the json dictionary.

to_json()
Convert the evidence object into a JSON dict.

4.1.5 Context (indra.statements.context)

class indra.statements.context.**BioContext**(*location=None*, *cell_line=None*, *cell_type=None*, *organ=None*, *disease=None*, *species=None*)

Bases: *indra.statements.context.Context*

An object representing the context of a Statement in biology.

Parameters

- **location** (*Optional[RefContext]*) – Cellular location, typically a sub-cellular compartment.
- **cell_line** (*Optional[RefContext]*) – Cell line context, e.g., a specific cell line, like BT20.
- **cell_type** (*Optional[RefContext]*) – Cell type context, broader than a cell line, like macrophage.
- **organ** (*Optional[RefContext]*) – Organ context.
- **disease** (*Optional[RefContext]*) – Disease context.
- **species** (*Optional[RefContext]*) – Species context.

class indra.statements.context.**Context**

Bases: object

An abstract class for Contexts.

class indra.statements.context.**MovementContext**(*locations=None*, *time=None*)

Bases: *indra.statements.context.Context*

An object representing the context of a movement between start and end points in time.

Parameters

- **locations** (*Optional[list[dict]*) – A list of dictionaries each containing a RefContext object representing geographical location context and its role (e.g. 'origin', 'destination', etc.)
- **time** (*Optional[TimeContext]*) – A TimeContext object representing the temporal context of the Statement.

class indra.statements.context.**RefContext**(*name=None*, *db_refs=None*)

Bases: object

An object representing a context with a name and references.

Parameters

- **name** (*Optional[str]*) – The name of the given context. In some cases a text name will not be available so this is an optional parameter with the default being None.
- **db_refs** (*Optional[dict]*) – A dictionary where each key is a namespace and each value is an identifier in that namespace, similar to the db_refs associated with Concepts/Agents.

class indra.statements.context.**TimeContext**(*text=None*, *start=None*, *end=None*, *duration=None*)

Bases: object

An object representing the time context of a Statement

Parameters

- **text** (*Optional[str]*) – A string representation of the time constraint, typically as seen in text.

- **start** (*Optional[datetime]*) – A *datetime* object representing the start time
- **end** (*Optional[datetime]*) – A *datetime* object representing the end time
- **duration** (*int*) – The duration of the time constraint in seconds

class indra.statements.context.**WorldContext**(*time=None*, *geo_location=None*)
Bases: *indra.statements.context.Context*

An object representing the context of a Statement in time and space.

Parameters

- **time** (*Optional[TimeContext]*) – A TimeContext object representing the temporal context of the Statement.
- **geo_location** (*Optional[RefContext]*) – The geographical location context represented as a RefContext

4.1.6 Input/output, serialization (indra.statements.io)

exception indra.statements.io.**InputError**
Bases: Exception

exception indra.statements.io.**UnresolvedUuidError**
Bases: Exception

indra.statements.io.**draw_stmt_graph**(*stmts*)
Render the attributes of a list of Statements as directed graphs.

The layout works well for a single Statement or a few Statements at a time. This function displays the plot of the graph using plt.show().

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of one or more INDRA Statements whose attribute graph should be drawn.

indra.statements.io.**pretty_print_stmts**(*stmt_list*, *stmt_limit=None*, *ev_limit=5*, *width=None*)
Print a formatted list of statements along with evidence text.

Requires the tabulate package (https://pypi.org/project/tabulate).

Parameters

- **stmt_list** (*List[Statement]*) – The list of INDRA Statements to be printed.
- **stmt_limit** (*Optional[int]*) – The maximum number of INDRA Statements to be printed. If None, all Statements are printed. (Default is None)
- **ev_limit** (*Optional[int]*) – The maximum number of Evidence to print for each Statement. If None, all evidence will be printed for each Statement. (Default is 5)
- **width** (*Optional[int]*) – Manually set the width of the table. If *None* the function will try to match the current terminal width using *os.get_terminal_size()*. If this fails the width defaults to 80 characters. The maximum width can be controlled by setting `pretty_print_max_width` using the *set_pretty_print_max_width()* function. This is useful in Jupyter notebooks where the environment returns a terminal size of 80 characters regardless of the width of the window. (Default is None).

Return type None

indra.statements.io.**print_stmt_summary**(*statements*)
Print a summary of a list of statements by statement type

Requires the tabulate package (https://pypi.org/project/tabulate).

Parameters **statements** (*List[Statement]*) – The list of INDRA Statements to be printed.

indra.statements.io.**set_pretty_print_max_width**(*new_max*)
Set the max display width for pretty prints, in characters.

indra.statements.io.**stmts_from_json**(*json_in*, *on_missing_support='handle'*)
Get a list of Statements from Statement jsons.

In the case of pre-assembled Statements which have *supports* and *supported_by* lists, the uuids will be replaced with references to Statement objects from the json, where possible. The method of handling missing support is controled by the *on_missing_support* key-word argument.

Parameters

- **json_in** (*iterable[dict]*) – A json list containing json dict representations of INDRA Statements, as produced by the *to_json* methods of subclasses of Statement, or equivalently by *stmts_to_json*.
- **on_missing_support** (*Optional[str]*) – Handles the behavior when a uuid reference in *supports* or *supported_by* attribute cannot be resolved. This happens because uuids can only be linked to Statements contained in the *json_in* list, and some may be missing if only some of all the Statements from pre- assembly are contained in the list.

  Options:

  - *'handle'* : (default) convert unresolved uuids into *Unresolved* Statement objects.
  - *'ignore'* : Simply omit any uuids that cannot be linked to any Statements in the list.
  - *'error'* : Raise an error upon hitting an un-linkable uuid.

Returns **stmts** – A list of INDRA Statements.

Return type list[Statement]

indra.statements.io.**stmts_from_json_file**(*fname*, *format='json'*)
Return a list of statements loaded from a JSON file.

Parameters

- **fname** (Union[str, Path, PathLike]) – Path to the JSON file to load statements from.
- **format** (*Optional[str]*) – One of 'json' to assume regular JSON formatting or 'jsonl' assuming each statement is on a new line.

Returns The list of INDRA Statements loaded from the JSOn file.

Return type list[indra.statements.Statement]

indra.statements.io.**stmts_to_json**(*stmts_in*, *use_sbo=False*, *matches_fun=None*)
Return the JSON-serialized form of one or more INDRA Statements.

Parameters

- **stmts_in** (*Statement or list[Statement]*) – A Statement or list of Statement objects to serialize into JSON.
- **use_sbo** (*Optional[bool]*) – If True, SBO annotations are added to each applicable element of the JSON. Default: False
- **matches_fun** (*Optional[function]*) – A custom function which, if provided, is used to construct the matches key which is then hashed and put into the return value. Default: None

Returns **json_dict** – JSON-serialized INDRA Statements.

Return type dict

indra.statements.io.**stmts_to_json_file**(*stmts*, *fname*, *format='json'*, ***kwargs*)
Serialize a list of INDRA Statements into a JSON file.

Parameters

- **stmts** (*list[indra.statement.Statements]*) – The list of INDRA Statements to serialize into the JSON file.
- **fname** (Union[str, Path, PathLike]) – Path to the JSON file to serialize Statements into.
- **format** (*Optional[str]*) – One of 'json' to use regular JSON with indent=1 formatting or 'jsonl' to put each statement on a new line without indents.

4.1.7 Validation (indra.statements.validate)

This module implements a number of functions that can be used to validate INDRA Statements. The available functions include ones that raise custom exceptions derived from ValueError if an invalidity is found. These come with a helpful error message that can be caught and printed to learn about the specific issue. Another set of functions do not raise exceptions, rather, return True or False depending on whether the given input is valid or invalid.

exception indra.statements.validate.**InvalidAgent**
Bases: ValueError

exception indra.statements.validate.**InvalidContext**
Bases: ValueError

exception indra.statements.validate.**InvalidIdentifier**
Bases: ValueError

Raised when the identifier doesn't match the pattern.

exception indra.statements.validate.**InvalidStatement**
Bases: ValueError

exception indra.statements.validate.**InvalidTextRefs**
Bases: ValueError

exception indra.statements.validate.**MissingIdentifier**
Bases: ValueError

Raised when the identifier is None.

exception indra.statements.validate.**UnknownIdentifier**
Bases: ValueError

Raise when the database is neither registered with identifiers.org or manually added to the indra.databases.identifiers.non_registry list.

exception indra.statements.validate.**UnknownNamespace**
Bases: ValueError

indra.statements.validate.**assert_valid_agent**(*agent*)
Raise InvalidAgent is there is an invalidity in the Agent.

Parameters **agent** (*indra.statements.Agent*) – The agent to check.

indra.statements.validate.**assert_valid_bio_context**(*context*)
Raise InvalidContext error if the given bio-context is invalid.

Parameters **context** (*indra.statements.BioContext*) – The context object to validate.

indra.statements.validate.**assert_valid_context**(*context*)
Raise InvalidContext error if the given context is invalid.

Parameters **context** (*indra.statements.Context*) – The context object to validate.

indra.statements.validate.**assert_valid_db_refs**(*db_refs*)
Raise InvalidIdentifier error if any of the entries in the given db_refs are invalid.

Parameters **db_refs** (*dict*) – A dict of database references, typically part of an INDRA Agent.

indra.statements.validate.**assert_valid_evidence**(*evidence*)
Raise an error if the given evidence is invalid.

Parameters **evidence** (*indra.statements.Evidence*) – The evidence object to validate.

indra.statements.validate.**assert_valid_id**(*db_ns*, *db_id*)
Raise InvalidIdentifier error if the ID is invalid in the given namespace.

Parameters

- **db_ns** (*str*) – The namespace.
- **db_id** (*str*) – The ID.

indra.statements.validate.**assert_valid_ns**(*db_ns*)
Raise UnknownNamespace error if the given namespace is unknown.

Parameters **db_ns** (*str*) – The namespace.

indra.statements.validate.**assert_valid_pmid_text_refs**(*evidence*)
Return True if the pmid attribute is consistent with text refs

indra.statements.validate.**assert_valid_statement**(*stmt*)
Raise an error if there is anything invalid in the given statement.

Parameters **stmt** (*indra.statements.Statement*) – An INDRA Statement to validate.

indra.statements.validate.**assert_valid_statement_semantics**(*stmt*)
Raise InvalidStatement error if the given statement is invalid.

Parameters **statement** (*indra.statements.Statement*) – The statement to check.

indra.statements.validate.**assert_valid_statements**(*stmts*)
Raise an error of any of the given statements is invalid.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of INDRA Statements to validate.

indra.statements.validate.**assert_valid_text_refs**(*text_refs*)
Raise an InvalidTextRefs error if the given text refs are invalid.

indra.statements.validate.**print_validation_report**(*stmts*)
Log the first validation error encountered for each given statement.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of INDRA Statements to validate.

indra.statements.validate.**validate_agent**(*agent*)
Return False if is there is an invalidity in the Agent, otherwise True.

Parameters **agent** (*indra.statements.Agent*) – The agent to check.

Returns True if the agent is valid, False otherwise.

Return type bool

indra.statements.validate.**validate_db_refs**(*db_refs*)
Return True if all the entries in the given db_refs are valid.

Parameters **db_refs** (*dict*) – A dict of database references, typically part of an INDRA Agent.

Returns True if all the entries are valid, else False.

Return type bool

indra.statements.validate.**validate_evidence**(*evidence*)
Return False if the given evidence is invalid, otherwise True.

Parameters **evidence** (*indra.statements.Evidence*) – The evidence object to validate.

Returns True if the evidence is valid, otherwise False.

Return type bool

indra.statements.validate.**validate_id**(*db_ns*, *db_id*)
Return True if the given ID is valid in the given namespace.

Parameters

- **db_ns** (*str*) – The namespace.
- **db_id** (*str*) – The ID.

Returns True if the given ID is valid in the given namespace.

Return type bool

indra.statements.validate.**validate_ns**(*db_ns*)
Return True if the given namespace is known.

Parameters **db_ns** (*str*) – The namespace.

Returns True if the given namepsace is known, otherwise False.

Return type bool

indra.statements.validate.**validate_statement**(*stmt*)
Return True if all the groundings in the given statement are valid.

Parameters **stmt** (*indra.statements.Statement*) – An INDRA Statement to validate.

Returns True if all the db_refs entries of the Agents in the given Statement are valid, else False.

Return type bool

indra.statements.validate.**validate_text_refs**(*text_refs*)
Return True if the given text refs are valid.

4.1.8 Resource access (indra.statements.resources)

exception indra.statements.resources.**InvalidLocationError**(*name*)
Bases: ValueError

Invalid cellular component name.

exception indra.statements.resources.**InvalidResidueError**(*name*)
Bases: ValueError

Invalid residue (amino acid) name.

indra.statements.resources.**get_valid_residue**(*residue*)
Check if the given string represents a valid amino acid residue.

4.1.9 Utils (indra.statements.util)

indra.statements.util.**make_hash**(*s*, *n_bytes*)
Make the hash from a matches key.

4.2 Processors for knowledge input (indra.sources)

INDRA interfaces with and draws knowledge from many sources including reading systems (some that extract biological mechanisms, and some that extract general causal interactions from text) and also from structured databases, which are typically human-curated or derived from experimental data.

4.2.1 Reading Systems

REACH (indra.sources.reach)

REACH is a biology-oriented machine reading system which uses a cascade of grammars to extract biological mechanisms from free text.

To cover a wide range of use cases and scenarios, there are currently 4 different ways in which INDRA can use REACH.

1. INDRA communicating with a locally running REACH Server (indra.sources.reach.api)

Setup and usage: Follow standard instructions to install SBT. Then clone REACH and run the REACH web server.

```
git clone https://github.com/clulab/reach.git
cd reach
sbt 'runMain org.clulab.reach.export.server.ApiServer'
```

Then read text by specifying the url parameter when using *indra.sources.reach.process_text*.

```
from indra.sources import reach
rp = reach.process_text('MEK binds ERK', url=reach.local_text_url)
```

One limitation here is that the REACH sever is configured by default to limit the input to 2048 characters. To change this, edit the file *export/src/main/resources/reference.conf* in your local reach clone folder and add

```
http {
  server {
  // ...
  parsing {
    max-uri-length = 256k
  }
  // ...
  }
}
```

to increase the character limit.

It is also possible to read NXML (string or file) and process the text of a paper given its PMC ID or PubMed ID using other API methods in *indra.sources.reach.api*. Note that *reach.local_nxml_url* needs to be used as *url* in case NXML content is being read.

Advantages:

- Does not require setting up the pyjnius Python-Java bridge.
- Does not require assembling a REACH JAR file.
- Allows local control the REACH version and configuration used to run the service.
- REACH is running in a separate process and therefore does not need to be initialized if a new Python session is started.

Disadvantages:

- First request might be time-consuming as REACH is loading additional resources.
- Only endpoints exposed by the REACH web server are available, i.e., no full object-level access to REACH components.

2. INDRA communicating with the UA REACH Server (`indra.sources.reach.api`)

Setup and usage: Does not require any additional setup after installing INDRA.

Read text using the default values for *offline* and *url* parameters.

```
from indra.sources import reach
rp = reach.process_text('MEK binds ERK')
```

It is also possible to read NXML (string or file) and process the content of a paper given its PMC ID or PubMed ID using other functions in *indra.sources.reach.api*.

Advantages:

- Does not require setting up the pyjnius Python-Java bridge.
- Does not require assembling a REACH JAR file or installing REACH at all locally.
- Suitable for initial prototyping or integration testing.

Disadvantages:

- Cannot handle high-throughput reading workflows due to limited server resources.
- No control over which REACH version is used to run the service.
- Difficulties processing NXML-formatted text (request times out) have been observed in the past.

3. INDRA using a REACH JAR through a Python-Java bridge (`indra.sources.reach.reader`)

Setup and usage:

Follow standard instructions for installing SBT. First, the REACH system and its dependencies need to be packaged as a fat JAR:

```
git clone https://github.com/clulab/reach.git
cd reach
sbt assembly
```

This creates a JAR file in reach/target/scala[version]/reach-[version].jar. Set the absolute path to this file on the REACHPATH environmental variable and then append REACHPATH to the CLASSPATH environmental variable (entries are separated by colons).

The *pyjnius* package needs to be set up and be operational. For more details, see *Pyjnius* setup instructions in the documentation.

Then, reading can be done using the *indra.sources.reach.process_text* function with the offline option.

```
from indra.sources import reach
rp = reach.process_text('MEK binds ERK', offline=True)
```

Other functions in *indra.sources.reach.api* can also be used with the offline option to invoke local, JAR-based reading.

Advantages:

- Doesn't require running a separate process for REACH and INDRA.
- Having a single REACH JAR file makes this solution easily portable.
- Through jnius, all classes in REACH become available for programmatic access.

Disadvantages:

- Requires configuring pyjnius which is often difficult (e.g., on Windows). Therefore this usage mode is generally not recommended.
- The ReachReader instance needs to be instantiated every time a new INDRA session is started which is time consuming.

4. Use REACH separately to produce output files and then process those with INDRA

In this usage mode REACH is not directly invoked by INDRA. Rather, REACH is set up and run independently of INDRA to produce output files for a set of text content. For more information on running REACH on a set of text or NXML files, see the REACH documentation at: https://github.com/clulab/reach. Note that INDRA uses the *fries* output format produced by REACH.

Once REACH output has been obtained in the *fries* JSON format, one can use *indra.sources.reach.api.process_json_file* in INDRA to process each JSON file.

REACH API (indra.sources.reach.api)

Methods for obtaining a reach processor containing indra statements.

Many file formats are supported. Many will run reach.

indra.sources.reach.api.**process_json_file**(*file_name*, *citation=None*, *organism_priority=None*)

Return a ReachProcessor by processing the given REACH json file.

The output from the REACH parser is in this json format. This function is useful if the output is saved as a file and needs to be processed. For more information on the format, see: https://github.com/clulab/reach

Parameters

- **file_name** (*str*) – The name of the json file to be processed.
- **citation** (*Optional[str]*) – A PubMed ID passed to be used in the evidence for the extracted INDRA Statements. Default: None
- **organism_priority** (*Optional[list of str]*) – A list of Taxonomy IDs providing prioritization among organisms when choosing protein grounding. If not given, the default behavior takes the first match produced by Reach, which is prioritized to be a human protein if such a match exists.

Returns **rp** – A ReachProcessor containing the extracted INDRA Statements in rp.statements.

Return type *ReachProcessor*

indra.sources.reach.api.**process_json_str**(*json_str*, *citation=None*, *organism_priority=None*)
Return a ReachProcessor by processing the given REACH json string.

The output from the REACH parser is in this json format. For more information on the format, see: https://github.com/clulab/reach

Parameters

- **json_str** (*str*) – The json string to be processed.
- **citation** (*Optional[str]*) – A PubMed ID passed to be used in the evidence for the extracted INDRA Statements. Default: None
- **organism_priority** (*Optional[list of str]*) – A list of Taxonomy IDs providing prioritization among organisms when choosing protein grounding. If not given, the default behavior takes the first match produced by Reach, which is prioritized to be a human protein if such a match exists.

Returns **rp** – A ReachProcessor containing the extracted INDRA Statements in rp.statements.

Return type *ReachProcessor*

indra.sources.reach.api.**process_nxml_file**(*file_name*, *citation=None*, *offline=False*, *url=None*, *output_fname='reach_output.json'*, *organism_priority=None*)
Return a ReachProcessor by processing the given NXML file.

NXML is the format used by PubmedCentral for papers in the open access subset.

Parameters

- **file_name** (*str*) – The name of the NXML file to be processed.
- **citation** (*Optional[str]*) – A PubMed ID passed to be used in the evidence for the extracted INDRA Statements. Default: None
- **offline** (*Optional[bool]*) – If set to True, the REACH system is run offline via a JAR file. Otherwise (by default) the web service is called. Default: False
- **url** (*Optional[str]*) – URL for a REACH web service instance, which is used for reading if provided. If not provided but offline is set to False (its default value), the Arizona REACH web service is called (http://agathon.sista.arizona.edu:8080/odinweb/api/help). Default: None
- **output_fname** (*Optional[str]*) – The file to output the REACH JSON output to. Defaults to reach_output.json in current working directory.
- **organism_priority** (*Optional[list of str]*) – A list of Taxonomy IDs providing prioritization among organisms when choosing protein grounding. If not given, the default behavior takes the first match produced by Reach, which is prioritized to be a human protein if such a match exists.

Returns **rp** – A ReachProcessor containing the extracted INDRA Statements in rp.statements.

Return type *ReachProcessor*

indra.sources.reach.api.**process_nxml_str**(*nxml_str*, *citation=None*, *offline=False*, *url=None*, *output_fname='reach_output.json'*, *organism_priority=None*)
Return a ReachProcessor by processing the given NXML string.

NXML is the format used by PubmedCentral for papers in the open access subset.

Parameters

- **nxml_str** (*str*) – The NXML string to be processed.
- **citation** (*Optional[str]*) – A PubMed ID passed to be used in the evidence for the extracted INDRA Statements. Default: None
- **offline** (*Optional[bool]*) – If set to True, the REACH system is run offline via a JAR file. Otherwise (by default) the web service is called. Default: False
- **url** (*Optional[str]*) – URL for a REACH web service instance, which is used for reading if provided. If not provided but offline is set to False (its default value), the Arizona REACH web service is called (http://agathon.sista.arizona.edu:8080/odinweb/api/help). Default: None
- **output_fname** (*Optional[str]*) – The file to output the REACH JSON output to. Defaults to reach_output.json in current working directory.
- **organism_priority** (*Optional[list of str]*) – A list of Taxonomy IDs providing prioritization among organisms when choosing protein grounding. If not given, the default behavior takes the first match produced by Reach, which is prioritized to be a human protein if such a match exists.

Returns **rp** – A ReachProcessor containing the extracted INDRA Statements in rp.statements.

Return type *ReachProcessor*

indra.sources.reach.api.**process_pmc**(*pmc_id*, *offline=False*, *url=None*, *output_fname='reach_output.json'*, *organism_priority=None*)

Return a ReachProcessor by processing a paper with a given PMC id.

Uses the PMC client to obtain the full text. If it's not available, None is returned.

Parameters

- **pmc_id** (*str*) – The ID of a PubmedCentral article. The string may start with PMC but passing just the ID also works. Examples: 8511698, PMC8511698 https://www.ncbi.nlm.nih.gov/pmc/
- **offline** (*Optional[bool]*) – If set to True, the REACH system is run offline via a JAR file. Otherwise (by default) the web service is called. Default: False
- **url** (*Optional[str]*) – URL for a REACH web service instance, which is used for reading if provided. If not provided but offline is set to False (its default value), the Arizona REACH web service is called (http://agathon.sista.arizona.edu:8080/odinweb/api/help). Default: None
- **output_fname** (*Optional[str]*) – The file to output the REACH JSON output to. Defaults to reach_output.json in current working directory.
- **organism_priority** (*Optional[list of str]*) – A list of Taxonomy IDs providing prioritization among organisms when choosing protein grounding. If not given, the default behavior takes the first match produced by Reach, which is prioritized to be a human protein if such a match exists.

Returns **rp** – A ReachProcessor containing the extracted INDRA Statements in rp.statements.

Return type *ReachProcessor*

indra.sources.reach.api.**process_pubmed_abstract**(*pubmed_id*, *offline=False*, *url=None*, *output_fname='reach_output.json'*, ***kwargs*)

Return a ReachProcessor by processing an abstract with a given Pubmed id.

Uses the Pubmed client to get the abstract. If that fails, None is returned.

Parameters

- **pubmed_id** (*str*) – The ID of a Pubmed article. The string may start with PMID but passing just the ID also works. Examples: 27168024, PMID27168024 https://www.ncbi.nlm.nih.gov/pubmed/
- **offline** (*Optional[bool]*) – If set to True, the REACH system is run offline via a JAR file. Otherwise (by default) the web service is called. Default: False
- **url** (*Optional[str]*) – URL for a REACH web service instance, which is used for reading if provided. If not provided but offline is set to False (its default value), the Arizona REACH web service is called (http://agathon.sista.arizona.edu:8080/odinweb/api/help). Default: None
- **output_fname** (*Optional[str]*) – The file to output the REACH JSON output to. Defaults to reach_output.json in current working directory.
- **organism_priority** (*Optional[list of str]*) – A list of Taxonomy IDs providing prioritization among organisms when choosing protein grounding. If not given, the default behavior takes the first match produced by Reach, which is prioritized to be a human protein if such a match exists.
- ****kwargs** (*keyword arguments*) – All other keyword arguments are passed directly to *process_text*.

Returns **rp** – A ReachProcessor containing the extracted INDRA Statements in rp.statements.

Return type *ReachProcessor*

indra.sources.reach.api.**process_text**(*text*, *citation=None*, *offline=False*, *url=None*, *output_fname='reach_output.json'*, *timeout=None*, *organism_priority=None*)

Return a ReachProcessor by processing the given text.

Parameters

- **text** (*str*) – The text to be processed.
- **citation** (*Optional[str]*) – A PubMed ID passed to be used in the evidence for the extracted INDRA Statements. This is used when the text to be processed comes from a publication that is not otherwise identified. Default: None
- **offline** (*Optional[bool]*) – If set to True, the REACH system is run offline via a JAR file. Otherwise (by default) the web service is called. Default: False
- **url** (*Optional[str]*) – URL for a REACH web service instance, which is used for reading if provided. If not provided but offline is set to False (its default value), the Arizona REACH web service is called (http://agathon.sista.arizona.edu:8080/odinweb/api/help). Default: None
- **output_fname** (*Optional[str]*) – The file to output the REACH JSON output to. Defaults to reach_output.json in current working directory.
- **timeout** (*Optional[float]*) – This only applies when reading online (*offline=False*). Only wait for *timeout* seconds for the api to respond.
- **organism_priority** (*Optional[list of str]*) – A list of Taxonomy IDs providing prioritization among organisms when choosing protein grounding. If not given, the default behavior takes the first match produced by Reach, which is prioritized to be a human protein if such a match exists.

Returns **rp** – A ReachProcessor containing the extracted INDRA Statements in rp.statements.

Return type *ReachProcessor*

REACH Processor (indra.sources.reach.processor)

class indra.sources.reach.processor.**ReachProcessor**(*json_dict*, *pmid=None*, *organism_priority=None*)
The ReachProcessor extracts INDRA Statements from REACH parser output.

Parameters

- **json_dict** (*dict*) – A JSON dictionary containing the REACH extractions.
- **pmid** (*Optional[str]*) – The PubMed ID associated with the extractions. This can be passed in case the PMID cannot be determined from the extractions alone.`

tree
The objectpath Tree object representing the extractions.

Type objectpath.Tree

statements
A list of INDRA Statements that were extracted by the processor.

Type list[indra.statements.Statement]

citation
The PubMed ID associated with the extractions.

Type str

all_events
The frame IDs of all events by type in the REACH extraction.

Type dict[str, str]

organism_priority
A list of Taxonomy IDs providing prioritization among organisms when choosing protein grounding. If not given, the default behavior takes the first match produced by Reach, which is prioritized to be a human protein if such a match exists.

Type list[str]

get_activation()
Extract INDRA Activation Statements.

get_all_events()
Gather all event IDs in the REACH output by type.

These IDs are stored in the self.all_events dict.

get_complexes()
Extract INDRA Complex Statements.

get_modifications()
Extract Modification INDRA Statements.

get_regulate_amounts()
Extract RegulateAmount INDRA Statements.

get_translocation()
Extract INDRA Translocation Statements.

print_event_statistics()
Print the number of events in the REACH output by type.

class indra.sources.reach.processor.**Site**(*residue*, *position*)

property position
Alias for field number 1

property residue
Alias for field number 0

indra.sources.reach.processor.**determine_reach_subtype**(*event_name*)
Returns the category of reach rule from the reach rule instance.

Looks at a list of regular expressions corresponding to reach rule types, and returns the longest regexp that matches, or None if none of them match.

Parameters **evidence** (*indra.statements.Evidence*) – A reach evidence object to subtype

Returns **best_match** – A regular expression corresponding to the reach rule that was used to extract this evidence

Return type str

indra.sources.reach.processor.**prioritize_organism_grounding**(*first_id*, *xrefs*, *organism_priority*)
Pick a prioritized organism-specific UniProt ID for a protein.

REACH reader (indra.sources.reach.reader)

exception indra.sources.reach.reader.**ReachOfflineReadingError**

class indra.sources.reach.reader.**ReachReader**
The ReachReader wraps a singleton instance of the REACH reader.

This allows calling the reader many times without having to wait for it to start up each time.

api_ruler
An instance of the REACH ApiRuler class (java object).

Type org.clulab.reach.apis.ApiRuler

get_api_ruler()
Return the existing reader if it exists or launch a new one.

Returns **api_ruler** – An instance of the REACH ApiRuler class (java object).

Return type org.clulab.reach.apis.ApiRuler

TRIPS (indra.sources.trips)

TRIPS API (indra.sources.trips.api)

indra.sources.trips.api.**process_text**(*text*, *save_xml_name='trips_output.xml'*, *save_xml_pretty=True*, *offline=False*, *service_endpoint='drum'*, *service_host=None*)
Return a TripsProcessor by processing text.

Parameters

- **text** (*str*) – The text to be processed.
- **save_xml_name** (*Optional[str]*) – The name of the file to save the returned TRIPS extraction knowledge base XML. Default: trips_output.xml

- **save_xml_pretty** (*Optional[bool]*) – If True, the saved XML is pretty-printed. Some third-party tools require non-pretty-printed XMLs which can be obtained by setting this to False. Default: True
- **offline** (*Optional[bool]*) – If True, offline reading is used with a local instance of DRUM, if available. Default: False
- **service_endpoint** (*Optional[str]*) – Selects the TRIPS/DRUM web service endpoint to use. Is a choice between "drum" (default) and "drum-dev", a nightly build.
- **service_host** (*Optional[str]*) – Address of a service host different from the public IHMC server (e.g., a locally running service).

Returns **tp** – A TripsProcessor containing the extracted INDRA Statements in tp.statements.

Return type *TripsProcessor*

indra.sources.trips.api.**process_xml**(*xml_string*)
Return a TripsProcessor by processing a TRIPS EKB XML string.

Parameters **xml_string** (*str*) – A TRIPS extraction knowledge base (EKB) string to be processed. http://trips.ihmc.us/parser/api.html

Returns **tp** – A TripsProcessor containing the extracted INDRA Statements in tp.statements.

Return type *TripsProcessor*

indra.sources.trips.api.**process_xml_file**(*file_name*)
Return a TripsProcessor by processing a TRIPS EKB XML file.

Parameters **file_name** (*str*) – Path to a TRIPS extraction knowledge base (EKB) file to be processed.

Returns **tp** – A TripsProcessor containing the extracted INDRA Statements in tp.statements.

Return type *TripsProcessor*

TRIPS Processor (indra.sources.trips.processor)

class indra.sources.trips.processor.**TripsProcessor**(*xml_string*)
The TripsProcessor extracts INDRA Statements from a TRIPS XML.

For more details on the TRIPS EKB XML format, see http://trips.ihmc.us/parser/cgi/drum

Parameters **xml_string** (*str*) – A TRIPS extraction knowledge base (EKB) in XML format as a string.

tree
An ElementTree object representation of the TRIPS EKB XML.

Type xml.etree.ElementTree.Element

statements
A list of INDRA Statements that were extracted from the EKB.

Type list[indra.statements.Statement]

doc_id
The PubMed ID of the paper that the extractions are from.

Type str

sentences
The list of all sentences in the EKB with their IDs

Type dict[str: str]

paragraphs
The list of all paragraphs in the EKB with their IDs

Type dict[str: str]

par_to_sec
A map from paragraph IDs to their associated section types

Type dict[str: str]

extracted_events
A list of Event elements that have been extracted as INDRA Statements.

Type list[xml.etree.ElementTree.Element]

get_activations()
Extract direct Activation INDRA Statements.

get_activations_causal()
Extract causal Activation INDRA Statements.

get_activations_stimulate()
Extract Activation INDRA Statements via stimulation.

get_active_forms()
Extract ActiveForm INDRA Statements.

get_active_forms_state()
Extract ActiveForm INDRA Statements.

get_agents()
Return list of INDRA Agents corresponding to TERMs in the EKB.

This is meant to be used when entities e.g. "phosphorylated ERK", rather than events need to be extracted from processed natural language. These entities with their respective states are represented as INDRA Agents.

Returns **agents** – List of INDRA Agents extracted from EKB.

Return type list[indra.statements.Agent]

get_all_events()
Make a list of all events in the TRIPS EKB.

The events are stored in self.all_events.

get_complexes()
Extract Complex INDRA Statements.

get_degradations()
Extract Degradation INDRA Statements.

get_modifications()
Extract all types of Modification INDRA Statements.

get_modifications_indirect()
Extract indirect Modification INDRA Statements.

get_regulate_amounts()
Extract Increase/DecreaseAmount Statements.

get_syntheses()
Extract IncreaseAmount INDRA Statements.

get_term_agents()
Return dict of INDRA Agents keyed by corresponding TERMs in the EKB.

This is meant to be used when entities e.g. "phosphorylated ERK", rather than events need to be extracted from processed natural language. These entities with their respective states are represented as INDRA Agents. Further, each key of the dictionary corresponds to the ID assigned by TRIPS to the given TERM that the Agent was extracted from.

Returns **agents** – Dict of INDRA Agents extracted from EKB.

Return type dict[str, indra.statements.Agent]

TRIPS Web-service Client (indra.sources.trips.client)

indra.sources.trips.client.**get_xml**(*html*, *content_tag='ekb'*, *fail_if_empty=False*)
Extract the content XML from the HTML output of the TRIPS web service.

Parameters

- **html** (*str*) – The HTML output from the TRIPS web service.
- **content_tag** (*str*) – The xml tag used to label the content. Default is 'ekb'.
- **fail_if_empty** (*bool*) – If True, and if the xml content found is an empty string, raise an exception. Default is False.

Returns

- *The extraction knowledge base (e.g. EKB) XML that contains the event and*
- *term extractions.*

indra.sources.trips.client.**save_xml**(*xml_str*, *file_name*, *pretty=True*)
Save the TRIPS EKB XML in a file.

Parameters

- **xml_str** (*str*) – The TRIPS EKB XML string to be saved.
- **file_name** (*str*) – The name of the file to save the result in.
- **pretty** (*Optional[bool]*) – If True, the XML is pretty printed.

indra.sources.trips.client.**send_query**(*text*, *service_endpoint='drum'*, *query_args=None*, *service_host=None*)
Send a query to the TRIPS web service.

Parameters

- **text** (*str*) – The text to be processed.
- **service_endpoint** (*Optional[str]*) – Selects the TRIPS/DRUM web service endpoint to use. Is a choice between "drum" (default), "drum-dev", a nightly build, and "cwms" for use with more general knowledge extraction.
- **query_args** (*Optional[dict]*) – A dictionary of arguments to be passed with the query.
- **service_host** (*Optional[str]*) – The server's base URL under which service_endpoint is an endpoint. By default, IHMC's public server is used.

Returns **html** – The HTML result returned by the web service.

Return type str

TRIPS/DRUM Local Reader (indra.sources.trips.drum_reader)

class indra.sources.trips.drum_reader.**DrumReader**(***kwargs*)

Agent which processes text through a local TRIPS/DRUM instance.

This class is implemented as a communicative agent which sends and receives KQML messages through a socket. It sends text (ideally in small blocks like one sentence at a time) to the running DRUM instance and receives extraction knowledge base (EKB) XML responses asynchronously through the socket. To install DRUM and its dependencies locally, follow instructions at: https://github.com/wdebeaum/drum Once installed, run *drum/bin/trips-drum -nouser* to run DRUM without a GUI. Once DRUM is running, this class can be instantiated as *dr = DrumReader()*, at which point it attempts to connect to DRUM via the socket. You can use *dr.read_text(text)* to send text for reading. In another usage more, *dr.read_pmc(pmcid)* can be used to read a full open-access PMC paper. Receiving responses can be started as *dr.start()* which waits for responses from the reader and returns when all responses were received. Once finished, the list of EKB XML extractions can be accessed via *dr.extractions*.

Parameters

- **run_drum** (*Optional[bool]*) – If True, the DRUM reading system is launched as a subprocess for reading. If False, DRUM is expected to be running independently. Default: False
- **drum_system** (*Optional[subproces.Popen]*) – A handle to the subprocess of a running DRUM system instance. This can be passed in in case the instance is to be reused rather than restarted. Default: None
- ****kwargs** – All other keyword arguments are passed through to the DrumReader KQML module's constructor.

extractions

A list of EKB XML extractions corresponding to the input text list.

Type list[str]

drum_system

A subprocess handle that points to a running instance of the DRUM reading system. In case the DRUM system is running independently, this is None.

Type subprocess.Popen

read_pmc(*pmcid*)

Read a given PMC article.

Parameters **pmcid** (*str*) – The PMC ID of the article to read. Note that only articles in the open-access subset of PMC will work.

read_text(*text*)

Read a given text phrase.

Parameters **text** (*str*) – The text to read. Typically a sentence or a paragraph.

receive_reply(*msg*, *content*)

Handle replies with reading results.

Sparser (indra.sources.sparser)

Sparser API (indra.sources.sparser.api)

Provides an API used to run and get Statements from the Sparser reading system.

indra.sources.sparser.api.**get_version**()
Return the version of the Sparser executable on the path.

Returns **version** – The version of Sparser that is found on the Sparser path.

Return type str

indra.sources.sparser.api.**make_nxml_from_text**(*text*)
Return raw text wrapped in NXML structure.

Parameters **text** (*str*) – The raw text content to be wrapped in an NXML structure.

Returns **nxml_str** – The NXML string wrapping the raw text input.

Return type str

indra.sources.sparser.api.**process_json_dict**(*json_dict*)
Return processor with Statements extracted from a Sparser JSON.

Parameters **json_dict** (*dict*) – The JSON object obtained by reading content with Sparser, using the 'json' output mode.

Returns **sp** – A SparserJSONProcessor which has extracted Statements as its statements attribute.

Return type SparserJSONProcessor

indra.sources.sparser.api.**process_nxml_file**(*fname*, *output_fmt='json'*, *outbuf=None*, *cleanup=True*, ***kwargs*)
Return processor with Statements extracted by reading an NXML file.

Parameters

- **fname** (*str*) – The path to the NXML file to be read.
- **output_fmt** (*Optional[str]*) – The output format to obtain from Sparser, with the two options being 'json' and 'xml'. Default: 'json'
- **outbuf** (*Optional[file]*) – A file like object that the Sparser output is written to.
- **cleanup** (*Optional[bool]*) – If True, the output file created by Sparser is removed. Default: True

Returns

- **sp** (*SparserXMLProcessor or SparserJSONProcessor depending on what output*)
- *format was chosen.*

indra.sources.sparser.api.**process_nxml_str**(*nxml_str*, *output_fmt='json'*, *outbuf=None*, *cleanup=True*, *key=''*, ***kwargs*)
Return processor with Statements extracted by reading an NXML string.

Parameters

- **nxml_str** (*str*) – The string value of the NXML-formatted paper to be read.
- **output_fmt** (*Optional[str]*) – The output format to obtain from Sparser, with the two options being 'json' and 'xml'. Default: 'json'
- **outbuf** (*Optional[file]*) – A file like object that the Sparser output is written to.

- **cleanup** (*Optional[bool]*) – If True, the temporary file created in this function, which is used as an input file for Sparser, as well as the output file created by Sparser are removed. Default: True
- **key** (*Optional[str]*) – A key which is embedded into the name of the temporary file passed to Sparser for reading. Default is empty string.

Returns

- *SparserXMLProcessor or SparserJSONProcessor depending on what output*
- *format was chosen.*

indra.sources.sparser.api.**process_sparser_output**(*output_fname*, *output_fmt='json'*)

Return a processor with Statements extracted from Sparser XML or JSON

Parameters

- **output_fname** (*str*) – The path to the Sparser output file to be processed. The file can either be JSON or XML output from Sparser, with the output_fmt parameter defining what format is assumed to be processed.
- **output_fmt** (*Optional[str]*) – The format of the Sparser output to be processed, can either be 'json' or 'xml'. Default: 'json'

Returns

- **sp** (*SparserXMLProcessor or SparserJSONProcessor depending on what output*)
- *format was chosen.*

indra.sources.sparser.api.**process_text**(*text*, *output_fmt='json'*, *outbuf=None*, *cleanup=True*, *key=''*, ***kwargs*)

Return processor with Statements extracted by reading text with Sparser.

Parameters

- **text** (*str*) – The text to be processed
- **output_fmt** (*Optional[str]*) – The output format to obtain from Sparser, with the two options being 'json' and 'xml'. Default: 'json'
- **outbuf** (*Optional[file]*) – A file like object that the Sparser output is written to.
- **cleanup** (*Optional[bool]*) – If True, the temporary file created, which is used as an input file for Sparser, as well as the output file created by Sparser are removed. Default: True
- **key** (*Optional[str]*) – A key which is embedded into the name of the temporary file passed to Sparser for reading. Default is empty string.

Returns

- *SparserXMLProcessor or SparserJSONProcessor depending on what output*
- *format was chosen.*

indra.sources.sparser.api.**process_xml**(*xml_str*)

Return processor with Statements extracted from a Sparser XML.

Parameters **xml_str** (*str*) – The XML string obtained by reading content with Sparser, using the 'xml' output mode.

Returns **sp** – A SparserXMLProcessor which has extracted Statements as its statements attribute.

Return type SparserXMLProcessor

indra.sources.sparser.api.**run_sparser**(*fname*, *output_fmt*, *outbuf=None*, *timeout=600*)
Return the path to reading output after running Sparser reading.

Parameters

- **fname** (*str*) – The path to an input file to be processed. Due to the Spaser executable's assumptions, the file name needs to start with PMC and should be an NXML formatted file.
- **output_fmt** (*Optional[str]*) – The format in which Sparser should produce its output, can either be 'json' or 'xml'.
- **outbuf** (*Optional[file]*) – A file like object that the Sparser output is written to.
- **timeout** (*int*) – The number of seconds to wait until giving up on this one reading. The default is 600 seconds (i.e. 10 minutes). Sparcer is a fast reader and the typical type to read a single full text is a matter of seconds.

Returns **output_path** – The path to the output file created by Sparser.

Return type str

MedScan (indra.sources.medscan)

MedScan is Elsevier's proprietary text-mining system for reading the biological literature. This INDRA module enables processing output files (in CSXML format) from the MedScan system into INDRA Statements.

MedScan API (indra.sources.medscan.api)

indra.sources.medscan.api.**process_directory**(*directory_name*, *lazy=False*)
Processes a directory filled with CSXML files, first normalizing the character encodings to utf-8, and then processing into a list of INDRA statements.

Parameters

- **directory_name** (*str*) – The name of a directory filled with csxml files to process
- **lazy** (*bool*) – If True, the statements will not be generated immediately, but rather a generator will be formulated, and statements can be retrieved by using *iter_statements*. If False, the *statements* attribute will be populated immediately. Default is False.

Returns **mp** – A MedscanProcessor populated with INDRA statements extracted from the csxml files

Return type *indra.sources.medscan.processor.MedscanProcessor*

indra.sources.medscan.api.**process_directory_statements_sorted_by_pmid**(*directory_name*)
Processes a directory filled with CSXML files, first normalizing the character encoding to utf-8, and then processing into INDRA statements sorted by pmid.

Parameters **directory_name** (*str*) – The name of a directory filled with csxml files to process

Returns **pmid_dict** – A dictionary mapping pmids to a list of statements corresponding to that pmid

Return type dict

indra.sources.medscan.api.**process_file**(*filename*, *interval=None*, *lazy=False*)
Process a CSXML file for its relevant information.

Consider running the fix_csxml_character_encoding.py script in indra/sources/medscan to fix any encoding issues in the input file before processing.

indra.sources.medscan.api.**filename**
The csxml file, containing Medscan XML, to process

Type str

indra.sources.medscan.api.**interval**
Select the interval of documents to read, starting with the `start`th document and ending before the `end`th document. If either is None, the value is considered undefined. If the value exceeds the bounds of available documents, it will simply be ignored.

Type (start, end) or None

indra.sources.medscan.api.**lazy**
If True, the statements will not be generated immediately, but rather a generator will be formulated, and statements can be retrieved by using *iter_statements*. If False, the *statements* attribute will be populated immediately. Default is False.

Type bool

Returns **mp** – A MedscanProcessor object containing extracted statements

Return type *MedscanProcessor*

indra.sources.medscan.api.**process_file_sorted_by_pmid**(*file_name*)
Processes a file and returns a dictionary mapping pmids to a list of statements corresponding to that pmid.

Parameters **file_name** (*str*) – A csxml file to process

Returns **s_dict** – Dictionary mapping pmids to a list of statements corresponding to that pmid

Return type dict

MedScan Processor (indra.sources.medscan.processor)

class indra.sources.medscan.processor.**MedscanEntity**(*name*, *urn*, *type*, *properties*, *ch_start*, *ch_end*)

property ch_end
Alias for field number 5

property ch_start
Alias for field number 4

property name
Alias for field number 0

property properties
Alias for field number 3

property type
Alias for field number 2

property urn
Alias for field number 1

class indra.sources.medscan.processor.**MedscanProcessor**
Processes Medscan data into INDRA statements.

The special StateEffect event conveys information about the binding site of a protein modification. Sometimes this is paired with additional event information in a seperate SVO. When we encounter a StateEffect, we don't process into an INDRA statement right away, but instead store the site information and use it if we encounter a ProtModification event within the same sentence.

statements
A list of extracted INDRA statements

Type list<str>

sentence_statements
A list of statements for the sentence we are currently processing. Deduplicated and added to the main statement list when we finish processing a sentence.

Type list<str>

num_entities
The total number of subject or object entities the processor attempted to resolve

Type int

num_entities_not_found
The number of subject or object IDs which could not be resolved by looking in the list of entities or tagged phrases.

Type int

last_site_info_in_sentence
Stored protein site info from the last StateEffect event within the sentence, allowing us to combine information from StateEffect and ProtModification events within a single sentence in a single INDRA statement. This is reset at the end of each sentence

Type SiteInfo

agent_from_entity(*relation*, *entity_id*)
Create a (potentially grounded) INDRA Agent object from a given Medscan entity describing the subject or object.

Uses helper functions to convert a Medscan URN to an INDRA db_refs grounding dictionary.

If the entity has properties indicating that it is a protein with a mutation or modification, then constructs the needed ModCondition or MutCondition.

Parameters

- **relation** (MedscanRelation) – The current relation being processed
- **entity_id** (*str*) – The ID of the entity to process

Returns **agent** – A potentially grounded INDRA agent representing this entity

Return type indra.statements.Agent

process_csxml_file(*filename*, *interval=None*, *lazy=False*)
Processes a filehandle to MedScan csxml input into INDRA statements.

The CSXML format consists of a top-level <*batch*> root element containing a series of <*doc*> (document) elements, in turn containing <*sec*> (section) elements, and in turn containing <*sent*> (sentence) elements.

Within the <*sent*> element, a series of additional elements appear in the following order:

- <*toks*>, which contains a tokenized form of the sentence in its text attribute
- <*textmods*>, which describes any preprocessing/normalization done to the underlying text
- <*match*> elements, each of which contains one of more <*entity*> elements, describing entities in the text with their identifiers. The local IDs of each entities are given in the *msid* attribute of this element; these IDs are then referenced in any subsequent SVO elements.

- <*svo*> elements, representing subject-verb-object triples. SVO elements with a *type* attribute of *CONTROL* represent normalized regulation relationships; they often represent the normalized extraction of the immediately preceding (but unnormalized SVO element). However, in some cases there can be a "CONTROL" SVO element without its parent immediately preceding it.

Parameters

- **filename** (*string*) – The path to a Medscan csxml file.
- **interval** (*(start, end) or None*) – Select the interval of documents to read, starting with the `start`th document and ending before the `end`th document. If either is None, the value is considered undefined. If the value exceeds the bounds of available documents, it will simply be ignored.
- **lazy** (*bool*) – If True, only create a generator which can be used by the *get_statements* method. If True, populate the statements list now.

process_relation(*relation*, *last_relation*)

Process a relation into an INDRA statement.

Parameters

- **relation** (MedscanRelation) – The relation to process (a CONTROL svo with normalized verb)
- **last_relation** (MedscanRelation) – The relation immediately proceding the relation to process within the same sentence, or None if there are no preceding relations within the same sentence. This proceeding relation, if available, will refer to the same interaction but with an unnormalized (potentially more specific) verb, and is used when processing protein modification events.

class indra.sources.medscan.processor.**MedscanProperty**(*type*, *name*, *urn*)

property name

Alias for field number 1

property type

Alias for field number 0

property urn

Alias for field number 2

class indra.sources.medscan.processor.**MedscanRelation**(*pmid*, *uri*, *sec*, *entities*, *tagged_sentence*, *subj*, *verb*, *obj*, *svo_type*)

A structure representing the information contained in a Medscan SVO xml element as well as associated entities and properties.

pmid

The URI of the current document (such as a PMID)

Type str

sec

The section of the document the relation occurs in

Type str

entities

A dictionary mapping entity IDs from the same sentence to MedscanEntity objects.

Type dict

tagged_sentence
The sentence from which the relation was extracted, with some tagged phrases and annotations.

Type str

subj
The entity ID of the subject

Type str

verb
The verb in the relationship between the subject and the object

Type str

obj
The entity ID of the object

Type str

svo_type
The type of SVO relationship (for example, CONTROL indicates that the verb is normalized)

Type str

class indra.sources.medscan.processor.**ProteinSiteInfo**(*site_text*, *object_text*)
Represent a site on a protein, extracted from a StateEffect event.

Parameters

- **site_text** (*str*) – The site as a string (ex. S22)
- **object_text** (*str*) – The protein being modified, as the string that appeared in the original sentence

get_sites()
Parse the site-text string and return a list of sites.

Returns **sites** – A list of position-residue pairs corresponding to the site-text

Return type list[*Site*]

indra.sources.medscan.processor.**normalize_medscan_name**(*name*)
Removes the "complex" and "complex complex" suffixes from a medscan agent name so that it better corresponds with the grounding map.

Parameters **name** (*str*) – The Medscan agent name

Returns **norm_name** – The Medscan agent name with the "complex" and "complex complex" suffixes removed.

Return type str

TEES (indra.sources.tees)

The TEES processor requires an installaton of TEES. To install TEES:

1. Clone the latest stable version of TEES using

   git clone https://github.com/jbjorne/TEES.git

2. Put this TEES cloned repository in one of these three places: the same directory as INDRA, your home directory, or ~/Downloads. If you put TEES in a location other than one of these three places, you will need to pass this directory to *indra.sources.tees.api.process_text* each time you call it.

3. Run configure.py within the TEES installation to install TEES dependencies.

TEES API (indra.sources.tees.api)

This module provides a simplified API for invoking the Turku Event Extraction System (TEES) on text and extracting INDRA statement from TEES output.

See publication: Jari Björne, Sofie Van Landeghem, Sampo Pyysalo, Tomoko Ohta, Filip Ginter, Yves Van de Peer, Sofia Ananiadou and Tapio Salakoski, PubMed-Scale Event Extraction for Post-Translational Modifications, Epigenetics and Protein Structural Relations. Proceedings of BioNLP 2012, pages 82-90, 2012.

indra.sources.tees.api.**extract_output**(*output_dir*)

Extract the text of the a1, a2, and sentence segmentation files from the TEES output directory. These files are located within a compressed archive.

Parameters **output_dir** (*str*) – Directory containing the output of the TEES system

Returns

- **a1_text** (*str*) – The text of the TEES a1 file (specifying the entities)
- **a2_text** (*str*) – The text of the TEES a2 file (specifying the event graph)
- **sentence_segmentations** (*str*) – The text of the XML file specifying the sentence segmentation

indra.sources.tees.api.**process_text**(*text*, *pmid=None*, *python2_path=None*)

Processes the specified plain text with TEES and converts output to supported INDRA statements. Check for the TEES installation is the TEES_PATH environment variable, and configuration file; if not found, checks candidate paths in tees_candidate_paths. Raises an exception if TEES cannot be found in any of these places.

Parameters

- **text** (*str*) – Plain text to process with TEES
- **pmid** (*str*) – The PMID from which the paper comes from, to be stored in the Evidence object of statements. Set to None if this is unspecified.
- **python2_path** (*str*) – TEES is only compatible with python 2. This processor invokes this external python 2 interpreter so that the processor can be run in either python 2 or python 3. If None, searches for an executible named python2 in the PATH environment variable.

Returns **tp** – A TEESProcessor object which contains a list of INDRA statements extracted from TEES extractions

Return type *TEESProcessor*

indra.sources.tees.api.**run_on_text**(*text*, *python2_path*)

Runs TEES on the given text in a temporary directory and returns a temporary directory with TEES output.

The caller should delete this directory when done with it. This function runs TEES and produces TEES output files but does not process TEES output into INDRA statements.

Parameters

- **text** (*str*) – Text from which to extract relationships
- **python2_path** (*str*) – The path to the python 2 interpreter

Returns **output_dir** – Temporary directory with TEES output. The caller should delete this directgory when done with it.

Return type str

TEES Processor (indra.sources.tees.processor)

This module takes the TEES parse graph generated by parse_tees and converts it into INDRA statements.

See publication: Jari Björne, Sofie Van Landeghem, Sampo Pyysalo, Tomoko Ohta, Filip Ginter, Yves Van de Peer, Sofia Ananiadou and Tapio Salakoski, PubMed-Scale Event Extraction for Post-Translational Modifications, Epigenetics and Protein Structural Relations. Proceedings of BioNLP 2012, pages 82-90, 2012.

class indra.sources.tees.processor.**TEESProcessor**(*a1_text*, *a2_text*, *sentence_segmentations*, *pmid*)

Converts the output of the TEES reader to INDRA statements.

Only extracts a subset of INDRA statements. Currently supported statements are: * Phosphorylation * Dephosphorylation * Binding * IncreaseAmount * DecreaseAmount

Parameters

- **a1_text** (*str*) – The TEES a1 output file, with entity information
- **a2_text** (*str*) – The TEES a2 output file, with the event graph
- **sentence_segmentations** (*str*) – The TEES sentence segmentation XML output
- **pmid** (*int*) – The pmid which the text comes from, or None if we don't want to specify at the moment. Stored in the Evidence object for each statement.

statements

A list of INDRA statements extracted from the provided text via TEES

Type list[indra.statements.Statement]

connected_subgraph(*node*)

Returns the subgraph containing the given node, its ancestors, and its descendants.

Parameters **node** (*str*) – We want to create the subgraph containing this node.

Returns **subgraph** – The subgraph containing the specified node.

Return type networkx.DiGraph

find_event_parent_with_event_child(*parent_name*, *child_name*)

Finds all event nodes (is_event node attribute is True) that are of the type parent_name, that have a child event node with the type child_name.

find_event_with_outgoing_edges(*event_name*, *desired_relations*)

Gets a list of event nodes with the specified event_name and outgoing edges annotated with each of the specified relations.

Parameters

- **event_name** (*str*) – Look for event nodes with this name
- **desired_relations** (*list[str]*) – Look for event nodes with outgoing edges annotated with each of these relations

Returns **event_nodes** – Event nodes that fit the desired criteria

Return type list[str]

general_node_label(*node*)

Used for debugging - gives a short text description of a graph node.

get_entity_text_for_relation(*node*, *relation*)

Looks for an edge from node to some other node, such that the edge is annotated with the given relation. If there exists such an edge, and the node at the other edge is an entity, return that entity's text. Otherwise, returns None.

get_related_node(*node*, *relation*)
Looks for an edge from node to some other node, such that the edge is annotated with the given relation. If there exists such an edge, returns the name of the node it points to. Otherwise, returns None.

node_has_edge_with_label(*node_name*, *edge_label*)
Looks for an edge from node_name to some other node with the specified label. Returns the node to which this edge points if it exists, or None if it doesn't.

Parameters

- **G** – The graph object
- **node_name** – Node that the edge starts at
- **edge_label** – The text in the relation property of the edge

node_to_evidence(*entity_node*, *is_direct*)
Computes an evidence object for a statement.

We assume that the entire event happens within a single statement, and get the text of the sentence by getting the text of the sentence containing the provided node that corresponds to one of the entities participanting in the event.

The Evidence's pmid is whatever was provided to the constructor (perhaps None), and the annotations are the subgraph containing the provided node, its ancestors, and its descendants.

print_parent_and_children_info(*node*)
Used for debugging - prints a short description of a a node, its children, its parents, and its parents' children.

process_binding_statements()
Looks for Binding events in the graph and extracts them into INDRA statements.

In particular, looks for a Binding event node with outgoing edges with relations Theme and Theme2 - the entities these edges point to are the two constituents of the Complex INDRA statement.

process_decrease_expression_amount()
Looks for Negative_Regulation events with a specified Cause and a Gene_Expression theme, and processes them into INDRA statements.

process_increase_expression_amount()
Looks for Positive_Regulation events with a specified Cause and a Gene_Expression theme, and processes them into INDRA statements.

process_phosphorylation_statements()
Looks for Phosphorylation events in the graph and extracts them into INDRA statements.

In particular, looks for a Positive_regulation event node with a child Phosphorylation event node.

If Positive_regulation has an outgoing Cause edge, that's the subject If Phosphorylation has an outgoing Theme edge, that's the object If Phosphorylation has an outgoing Site edge, that's the site

indra.sources.tees.processor.**s2a**(*s*)
Makes an Agent from a string describing the agent.

ISI (indra.sources.isi)

This module provides an input interface and processor to the ISI reading system.

The reader is set up to run within a Docker container. For the ISI reader to run, set the Docker memory and swap space to the maximum.

ISI API (indra.sources.isi.api)

indra.sources.isi.api.**process_json_file**(*file_path*, *pmid=None*, *extra_annotations=None*, *add_grounding=True*, *molecular_complexes_only=False*)

Extracts statements from the given ISI output file.

Parameters

- **file_path** (*str*) – The ISI output file from which to extract statements
- **pmid** (*int*) – The PMID of the document being preprocessed, or None if not specified
- **extra_annotations** (*dict*) – Extra annotations to be added to each statement from this document (can be the empty dictionary)
- **add_grounding** (*Optional[bool]*) – If True the extracted Statements' grounding is mapped
- **molecular_complexes_only** (*Optional[bool]*) – If True, only Complex statements between molecular entities are retained after grounding.

indra.sources.isi.api.**process_nxml**(*nxml_filename*, *pmid=None*, *extra_annotations=None*, ***kwargs*)

Process an NXML file using the ISI reader

First converts NXML to plain text and preprocesses it, then runs the ISI reader, and processes the output to extract INDRA Statements.

Parameters

- **nxml_filename** (*str*) – nxml file to process
- **pmid** (*Optional[str]*) – pmid of this nxml file, to be added to the Evidence object of the extracted INDRA statements
- **extra_annotations** (*Optional[dict]*) – Additional annotations to add to the Evidence object of all extracted INDRA statements. Extra annotations called 'interaction' are ignored since this is used by the processor to store the corresponding raw ISI output.
- **num_processes** (*Optional[int]*) – Number of processes to parallelize over
- **cleanup** (*Optional[bool]*) – If True, the temporary folders created for preprocessed reading input and output are removed. Default: True
- **add_grounding** (*Optional[bool]*) – If True the extracted Statements' grounding is mapped
- **molecular_complexes_only** (*Optional[bool]*) – If True, only Complex statements between molecular entities are retained after grounding.

Returns **ip** – A processor containing extracted Statements

Return type *indra.sources.isi.processor.IsiProcessor*

indra.sources.isi.api.**process_output_folder**(*folder_path*, *pmids=None*, *extra_annotations=None*, *add_grounding=True*, *molecular_complexes_only=False*)

Recursively extracts statements from all ISI output files in the given directory and subdirectories.

Parameters

- **folder_path** (*str*) – The directory to traverse
- **pmids** (*Optional[str]*) – PMID mapping to be added to the Evidence of the extracted INDRA Statements
- **extra_annotations** (*Optional[dict]*) – Additional annotations to add to the Evidence object of all extracted INDRA statements. Extra annotations called 'interaction' are ignored since this is used by the processor to store the corresponding raw ISI output.
- **add_grounding** (*Optional[bool]*) – If True the extracted Statements' grounding is mapped
- **molecular_complexes_only** (*Optional[bool]*) – If True, only Complex statements between molecular entities are retained after grounding.

indra.sources.isi.api.**process_preprocessed**(*isi_preprocessor*, *num_processes=1*, *output_dir=None*, *cleanup=True*, *add_grounding=True*, *molecular_complexes_only=False*)

Process a directory of abstracts and/or papers preprocessed using the specified IsiPreprocessor, to produce a list of extracted INDRA statements.

Parameters

- **isi_preprocessor** (*indra.sources.isi.preprocessor.IsiPreprocessor*) – Preprocessor object that has already preprocessed the documents we want to read and process with the ISI reader
- **num_processes** (*Optional[int]*) – Number of processes to parallelize over
- **output_dir** (*Optional[str]*) – The directory into which to put reader output; if omitted or None, uses a temporary directory.
- **cleanup** (*Optional[bool]*) – If True, the temporary folders created for preprocessed reading input and output are removed. Default: True
- **add_grounding** (*Optional[bool]*) – If True the extracted Statements' grounding is mapped
- **molecular_complexes_only** (*Optional[bool]*) – If True, only Complex statements between molecular entities are retained after grounding.

Returns **ip** – A processor containing extracted statements

Return type *indra.sources.isi.processor.IsiProcessor*

indra.sources.isi.api.**process_text**(*text*, *pmid=None*, ***kwargs*)

Process a string using the ISI reader and extract INDRA statements.

Parameters

- **text** (*str*) – A text string to process
- **pmid** (*Optional[str]*) – The PMID associated with this text (or None if not specified)
- **num_processes** (*Optional[int]*) – Number of processes to parallelize over
- **cleanup** (*Optional[bool]*) – If True, the temporary folders created for preprocessed reading input and output are removed. Default: True
- **add_grounding** (*Optional[bool]*) – If True the extracted Statements' grounding is mapped

- **molecular_complexes_only** (*Optional[bool]*) – If True, only Complex statements between molecular entities are retained after grounding.

Returns **ip** – A processor containing statements

Return type *indra.sources.isi.processor.IsiProcessor*

ISI Processor (indra.sources.isi.processor)

class indra.sources.isi.processor.**IsiProcessor**(*reader_output*, *pmid=None*, *extra_annotations=None*, *add_grounding=False*)

Processes the output of the ISI reader.

Parameters

- **reader_output** (*json*) – The output JSON of the ISI reader as a json object.
- **pmid** (*Optional[str]*) – The PMID to assign to the extracted Statements
- **extra_annotations** (*Optional[dict]*) – Annotations to be included with each extracted Statement
- **add_grounding** (*Optional[bool]*) – If True, Gilda is used as a service to ground the Agents in the extracted Statements.

verbs
A list of verbs that have appeared in the processed ISI output

Type set[str]

statements
Extracted statements

Type list[indra.statements.Statement]

get_statements()
Process reader output to produce INDRA Statements.

retain_molecular_complexes()
Filter the statements to Complexes between molecular entities.

Geneways (indra.sources.geneways)

Geneways API (indra.sources.geneways.api)

This module provides a simplified API for invoking the Geneways input processor , which converts extracted information collected with Geneways into INDRA statements.

See publication: Rzhetsky, Andrey, Ivan Iossifov, Tomohiro Koike, Michael Krauthammer, Pauline Kra, Mitzi Morris, Hong Yu et al. "GeneWays: a system for extracting, analyzing, visualizing, and integrating molecular pathway data." Journal of biomedical informatics 37, no. 1 (2004): 43-53.

indra.sources.geneways.api.**process_geneways_files**(*input_folder='/home/docs/checkouts/readthedocs.org/user_builds/indr*, *get_evidence=True*)

Reads in Geneways data and returns a list of statements.

Parameters

- **input_folder** (*Optional[str]*) – A folder in which to search for Geneways data. Looks for these Geneways extraction data files: human_action.txt, human_actionmention.txt, human_symbols.txt. Omit this parameter to use the default input folder which is indra/data.
- **get_evidence** (*Optional[bool]*) – Attempt to find the evidence text for an extraction by downloading the corresponding text content and searching for the given offset in the text to get the evidence sentence. Default: True

Returns **gp** – A GenewaysProcessor object which contains a list of INDRA statements generated from the Geneways action mentions.

Return type *GenewaysProcessor*

Geneways Processor (indra.sources.geneways.processor)

This module provides an input processor for information extracted using the Geneways software suite, converting extraction data in Geneways format into INDRA statements.

See publication: Rzhetsky, Andrey, Ivan Iossifov, Tomohiro Koike, Michael Krauthammer, Pauline Kra, Mitzi Morris, Hong Yu et al. "GeneWays: a system for extracting, analyzing, visualizing, and integrating molecular pathway data." Journal of biomedical informatics 37, no. 1 (2004): 43-53.

class indra.sources.geneways.processor.**GenewaysProcessor**(*search_path*, *get_evidence=True*)

The GenewaysProcessors converts extracted Geneways action mentions into INDRA statements.

Parameters **search_path** (*list[str]*) – A list of directories in which to search for Geneways data

statements

A list of INDRA statements converted from Geneways action mentions, populated by calling the constructor

Type list[indra.statements.Statement]

make_statement(*action*, *mention*)

Makes an INDRA statement from a Geneways action and action mention.

Parameters

- **action** (*GenewaysAction*) – The mechanism that the Geneways mention maps to. Note that several text mentions can correspond to the same action if they are referring to the same relationship - there may be multiple Geneways action mentions corresponding to each action.
- **mention** (*GenewaysActionMention*) – The Geneways action mention object corresponding to a single mention of a mechanism in a specific text. We make a new INDRA statement corresponding to each action mention.

Returns **statement** – An INDRA statement corresponding to the provided Geneways action mention, or None if the action mention's type does not map onto any INDRA statement type in geneways_action_type_mapper.

Return type indra.statements.Statement

indra.sources.geneways.processor.**geneways_action_to_indra_statement_type**(*actiontype*, *plo*)

Return INDRA Statement corresponding to Geneways action type.

Parameters

- **actiontype** (*str*) – The verb extracted by the Geneways processor
- **plo** (*str*) – A one character string designating whether Geneways classifies this verb as a physical, logical, or other interaction

Returns If there is no mapping to INDRA statements from this action type the return value is None. If there is such a mapping, statement_generator is an anonymous function that takes in the subject agent, object agent, and evidence, in that order, and returns an INDRA statement object.

Return type statement_generator

RLIMS-P (indra.sources.rlimsp)

RLIMS-P is a rule-based reading system which extracts phosphorylation relationships with sites from text. RLIMS-P exposes a web service to submit PubMed IDs and PMC IDs for processing.

See also: https://research.bioinformatics.udel.edu/rlimsp/ and https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4568560/

RLIMS-P API (indra.sources.rlimsp.api)

indra.sources.rlimsp.api.**process_from_json_file**(*filename*, *doc_id_type=None*)
Process RLIMSP extractions from a bulk-download JSON file.

Parameters

- **filename** (*str*) – Path to the JSON file.
- **doc_id_type** (*Optional[str]*) – In some cases the RLIMS-P paragraph info doesn't contain 'pmid' or 'pmcid' explicitly, instead if contains a 'docId' key. This parameter allows defining what ID type 'docId' sould be interpreted as. Its values should be 'pmid' or 'pmcid' or None if not used.

Returns An RlimspProcessor which contains a list of extracted INDRA Statements in its statements attribute.

Return type *indra.sources.rlimsp.processor.RlimspProcessor*

indra.sources.rlimsp.api.**process_from_jsonish_str**(*jsonish_str*, *doc_id_type=None*)
Process RLIMSP extractions from a bulk-download JSON file.

Parameters

- **jsonish_str** (*str*) – The contents of one of the not-quite-json files you can find here: https://hershey.dbi.udel.edu/textmining/export
- **doc_id_type** (*Optional[str]*) – In some cases the RLIMS-P paragraph info doesn't contain 'pmid' or 'pmcid' explicitly, instead if contains a 'docId' key. This parameter allows defining what ID type 'docId' sould be interpreted as. Its values should be 'pmid' or 'pmcid' or None if not used.

Returns An RlimspProcessor which contains a list of extracted INDRA Statements in its statements attribute.

Return type *indra.sources.rlimsp.processor.RlimspProcessor*

indra.sources.rlimsp.api.**process_from_webservice**(*id_val*, *id_type='pmcid'*, *source='pmc'*)
Return an output from RLIMS-p for the given PubMed ID or PMC ID.

The web service is documented at: https://research.bioinformatics.udel.edu/itextmine/api/. The /data/rlims URL endpoint is extended with three additional elements: /{collection}/{key}/{value} where collection is "medline" or "pmc", key is "pmid" or "pmcid", and value is a specific PMID or PMCID.

Parameters

- **id_val** (*str*) – A PMCID, with the prefix PMC, or PMID, with no prefix, of the paper to be "read". Corresponds to the "value" argument of the REST API.
- **id_type** (*Optional[str]*) – Either 'pmid' or 'pmcid'. The default is 'pmcid'. Corresponds to the "key" argument of the REST API.
- **source** (*Optional[str]*) – Either 'pmc' or 'medline', whether you want pmc fulltext or medline abstracts. Corresponds to the "collection" argument of the REST API.

Returns An RlimspProcessor which contains a list of extracted INDRA Statements in its statements attribute.

Return type *indra.sources.rlimsp.processor.RlimspProcessor*

RLIMSP-P Processor (indra.sources.rlimsp.processor)

class indra.sources.rlimsp.processor.**RlimspParagraph**(*p_info*, *doc_id_type*)
An object that represents a single RLIMS-P Paragraph.

class indra.sources.rlimsp.processor.**RlimspProcessor**(*rlimsp_json*, *doc_id_type=None*)
Convert RLIMS-P JSON into INDRA Statements.

extract_statements()
Extract the statements from the json.

indra.sources.rlimsp.processor.**get_agent_from_entity_info**(*entity_info*)
Return an INDRA Agent by processing an entity_info dict.

Eidos (indra.sources.eidos)

Eidos is an open-domain machine reading system which uses a cascade of grammars to extract causal events from free text. It is ideal for modeling applications that are not specific to a given domain like molecular biology.

To cover a wide range of use cases and scenarios, there are currently 5 different ways in which INDRA can use Eidos.

In all cases for Eidos to provide grounding information to be included in INDRA Statements, it needs to be configured explicitly to do so. Please follow instructions at https://github.com/clulab/eidos#configuring to download and configure Eidos grounding resources.

1. INDRA communicating with a separately running Eidos webapp (indra.sources.eidos.client)

Setup and usage: Clone and run the Eidos web server.

```
git clone https://github.com/clulab/eidos.git
cd eidos
sbt webapp/run
```

Then read text by specifying the webserver parameter when using *indra.sources.eidos.process_text*.

```
from indra.sources import eidos
ep = eidos.process_text('rainfall causes floods',
                        webservice='http://localhost:9000')
```

Advantages:

- Does not require setting up the pyjnius Python-Java bridge

- Does not require assembling an Eidos JAR file

Disadvantages:

- Not all Eidos functionalities are immediately exposed through its webapp.

2. INDRA using an Eidos JAR directly through a Python-Java bridge (`indra.sources.eidos.reader`)

Setup and usage:

First, the Eidos system and its dependencies need to be packaged as a fat JAR:

```
git clone https://github.com/clulab/eidos.git
cd eidos
sbt assembly
```

This creates a JAR file in eidos/target/scala[version]/eidos-[version].jar. Set the absolute path to this file on the EIDOSPATH environmental variable and then append EIDOSPATH to the CLASSPATH environmental variable (entries are separated by colons).

The *pyjnius* package needs to be set up and be operational. For more details, see *Pyjnius* setup instructions in the documentation.

Then, reading can be done simply using the *indra.sources.eidos.process_text* function.

```
from indra.sources import eidos
ep = eidos.process_text('rainfall causes floods')
```

Advantages:

- Doesn't require running a separate process for Eidos and INDRA
- Having a single Eidos JAR file makes this solution portable

Disadvantages:

- Requires configuring pyjnius which is often difficult
- Requires building a large Eidos JAR file which can be time consuming
- The EidosReader instance needs to be instantiated every time a new INDRA session is started which is time consuming.

3. INDRA using a Flask sever wrapping an Eidos JAR in a separate process (`indra.sources.eidos.server`)

Setup and usage: Requires building an Eidos JAR and setting up pyjnius – see above.

First, run the server using

```
python -m indra.sources.eidos.server
```

Then point to the running server with the webservice parameter when calling *indra.sources.eidos.process_text*.

```
from indra.sources import eidos
ep = eidos.process_text('rainfall causes floods',
                        webservice='http://localhost:6666')
```

Advantages:

- EidosReader is instantiated by the Flask server in a separate process, therefore it isn't reloaded each time a new INDRA session is started
- Having a single Eidos JAR file makes this solution portable

Disadvantages:

- Currently does not offer any additional functionality compared to running the Eidos webapp directly
- Requires configuring pyjnius which is often difficult
- Requires building a large Eidos JAR file which can be time consuming

4. INDRA calling the Eidos CLI using java through the command line (`indra.sources.eidos.cli`)

Setup and usage: Requires building an Eidos JAR and setting EIDOSPATH but does not require setting up pyjnius – see above. To use, call any of the functions exposed in *indra.sources.eidos.cli*.

Advantages:

- Provides a Python-interface for running Eidos on "large scale" jobs, e.g., a large number of input files.
- Does not require setting up pyjnius since it uses Eidos via the command line.
- Provides a way to use any available entrypoint of Eidos.

Disadvantages:

- Requires building an Eidos JAR which can be time consuming.

5. Use Eidos separately to produce output files and then process those with INDRA

In this usage mode Eidos is not directly invoked by INDRA. Rather, Eidos is set up and run idenpendently of INDRA to produce JSON-LD output files for a set of text content. One can then use `indra.sources.eidos.api.process_json_file` in INDRA to process the JSON-LD output files.

Eidos API (`indra.sources.eidos.api`)

`indra.sources.eidos.api.`**`initialize_reader`**`()`
Instantiate an Eidos reader for fast subsequent reading.

`indra.sources.eidos.api.`**`process_json_bio`**`(`*json_dict*, *grounder=None*`)`
Return EidosProcessor with grounded Activation/Inhibition statements.

Parameters

- **json_dict** (*dict*) – The JSON-LD dict to be processed.
- **grounder** (*Optional[function]*) – A function which takes a text and an optional context as argument and returns a dict of groundings.

Returns **ep** – A EidosProcessor containing the extracted INDRA Statements in its statements attribute.

Return type *EidosProcessor*

`indra.sources.eidos.api.`**`process_json_bio_entities`**`(`*json_dict*, *grounder=None*`)`
Return INDRA Agents grounded to biological ontologies extracted from Eidos JSON-LD.

Parameters

- **json_dict** (*dict*) – The JSON-LD dict to be processed.
- **grounder** (*Optional[function]*) – A function which takes a text and an optional context as argument and returns a dict of groundings.

Returns A list of INDRA Agents which are derived from concepts extracted by Eidos from text.

Return type list of indra.statements.Agent

indra.sources.eidos.api.**process_text_bio**(*text*, *save_json='eidos_output.json'*, *webservice=None*, *grounder=None*)

Return an EidosProcessor by processing the given text.

This constructs a reader object via Java and extracts mentions from the text. It then serializes the mentions into JSON and processes the result with process_json.

Parameters

- **text** (*str*) – The text to be processed.
- **save_json** (*Optional[str]*) – The name of a file in which to dump the JSON output of Eidos.
- **webservice** (*Optional[str]*) – An Eidos reader web service URL to send the request to. If None, the reading is assumed to be done with the Eidos JAR rather than via a web service. Default: None
- **grounder** (*Optional[function]*) – A function which takes a text and an optional context as argument and returns a dict of groundings.

Returns **ep** – An EidosProcessor containing the extracted INDRA Statements in its statements attribute.

Return type *EidosProcessor*

indra.sources.eidos.api.**process_text_bio_entities**(*text*, *webservice=None*, *grounder=None*)

Return INDRA Agents grounded to biological ontologies extracted from text.

Parameters

- **text** (*str*) – Text to be processed.
- **webservice** (*Optional[str]*) – An Eidos reader web service URL to send the request to. If None, the reading is assumed to be done with the Eidos JAR rather than via a web service. Default: None
- **grounder** (*Optional[function]*) – A function which takes a text and an optional context as argument and returns a dict of groundings.

Returns A list of INDRA Agents which are derived from concepts extracted by Eidos from text.

Return type list of indra.statements.Agent

Eidos Processor (indra.sources.eidos.processor)

class indra.sources.eidos.processor.**EidosProcessor**(*json_dict*)
This processor extracts INDRA Statements from Eidos JSON-LD output.

Parameters **json_dict** (*dict*) – A JSON dictionary containing the Eidos extractions in JSON-LD format.

statements
A list of INDRA Statements that were extracted by the processor.

Type list[indra.statements.Statement]

extract_all_events()
Extract all events, including ones that are arguments of other statements.

The goal of this method is to extract events as standalone statements with their own dedicated evidence. This is different from the get_all_events method in that it extracts the event-specific evidence for each Event statement instead of propagating causal relation evidence into the Event after initial extraction.

extract_causal_relations()
Extract causal relations as Statements.

extract_correlations()
Extract correlations as Assocation statements.

extract_events()
Extract Events that are not arguments of other statements.

get_all_events()
Return a list of all standalone events from the existing list of extracted statements.

Note that this method only operates on statements already extracted into the processor's statements attribute. Note also that the evidences for events created from Influences and Assocations here are propagated from those statements; they are not equivalent to the original evidences for the events themselves (see extract_all_events method).

Returns **events** – A list of Events from original Events, and unrolled from Influences and Associations.

Return type list[indra.statements.Event]

get_concept(*entity*)
Return Concept from an Eidos entity.

get_evidence(*relation*)
Return the Evidence object for the INDRA Statment.

get_groundings(*entity*)
Return groundings as db_refs for an entity.

static get_hedging(*event*)
Return hedging markers attached to an event.

Example: "states": [{"@type": "State", "type": "HEDGE", "text": "could"}

static get_negation(*event*)
Return negation attached to an event.

Example: "states": [{"@type": "State", "type": "NEGATION", "text": "n't"}]

indra.sources.eidos.processor.**find_arg**(*event*, *arg_type*)
Return ID of the first argument of a given type

indra.sources.eidos.processor.**find_args**(*event*, *arg_type*)
Return IDs of all arguments of a given type

Eidos Bio Processor (indra.sources.eidos.bio_processor)

class indra.sources.eidos.bio_processor.**EidosBioProcessor**(*json_dict*, *grounder=None*)
Class to extract biology-oriented INDRA statements from Eidos output in a way that agents are grounded to biomedical ontologies.

Eidos Client (indra.sources.eidos.client)

indra.sources.eidos.client.**process_text**(*text*, *webservice*)
Process a given text with an Eidos webservice at the given address.

Note that in most cases this function should not be used directly, rather, used indirectly by calling *indra.sources.eidos.process_text* with the webservice parameter.

Parameters

- **text** (*str*) – The text to be read using Eidos.
- **webservice** (*str*) – The address where the Eidos web service is running, e.g., http://localhost:9000.

Returns A JSON dict of the results from the Eidos webservice.

Return type dict

Eidos Reader (indra.sources.eidos.reader)

class indra.sources.eidos.reader.**EidosReader**
Reader object keeping an instance of the Eidos reader as a singleton.

This allows the Eidos reader to need initialization when the first piece of text is read, the subsequent readings are done with the same instance of the reader and are therefore faster.

eidos_reader
A Scala object, an instance of the Eidos reading system. It is instantiated only when first processing text.

Type org.clulab.wm.eidos.EidosSystem

initialize_reader()
Instantiate the Eidos reader attribute of this reader.

process_text(*text*)
Return a mentions JSON object given text.

Parameters **text** (*str*) – Text to be processed.

Returns **json_dict** – A JSON object of mentions extracted from text.

Return type dict

Eidos Webserver (indra.sources.eidos.server)

This is a Python-based web server that can be run to read with Eidos. To run the server, do

    python -m indra.sources.eidos.server

and then submit POST requests to the *localhost:5000/process_text* endpoint with JSON content as *{'text': 'text to read'}*. The response will be the Eidos JSON-LD output. Another endpoint for regrounding entity texts is also available on the *reground* endpoint.

Eidos CLI (indra.sources.eidos.cli)

This is a Python based command line interface to Eidos to complement the Python-Java bridge based interface. EIDOSPATH (in the INDRA config.ini or as an environmental variable) needs to be pointing to a fat JAR of the Eidos system.

indra.sources.eidos.cli.**extract_and_process**(*path_in*, *path_out*, *process_fun*)
Run Eidos on a set of text files and process output with INDRA.

The output is produced in the specified output folder but the output files aren't processed by this function.

Parameters

- **path_in** (*str*) – Path to an input folder with some text files
- **path_out** (*str*) – Path to an output folder in which Eidos places the output JSON-LD files
- **process_fun** (*function*) – A function that takes a JSON dict as argument and returns an EidosProcessor.

Returns **stmts** – A list of INDRA Statements

Return type list[indra.statements.Statements]

indra.sources.eidos.cli.**extract_from_directory**(*path_in*, *path_out*)
Run Eidos on a set of text files in a folder.

The output is produced in the specified output folder but the output files aren't processed by this function.

Parameters

- **path_in** (*str*) – Path to an input folder with some text files
- **path_out** (*str*) – Path to an output folder in which Eidos places the output JSON-LD files

indra.sources.eidos.cli.**run_eidos**(*endpoint*, **args*)
Run a given enpoint of Eidos through the command line.

Parameters

- **endpoint** (*str*) – The class within the Eidos package to run, for instance 'apps.extract.ExtractFromDirectory' will run 'org.clulab.wm.eidos.apps.extract.ExtractFromDirectory'
- ***args** – Any further arguments to be passed as inputs to the class being run.

GNBR (indra.sources.gnbr)

This module extracts INDRA Statements from the Global Network of Biomedical Relationships resource

GNBR API (indra.sources.gnbr.api)

indra.sources.gnbr.api.**process_chemical_disease**(*part1_path*, *part2_path*, *indicator_only=True*)
Process chemical–disease interactions.

Parameters

- **part1_path** (str) – Path to the first dataset which contains dependency paths and themes.
- **part2_path** (str) – Path to the second dataset which contains dependency paths and entity pairs.
- **indicator_only** (bool) – A switch to filter the data which is part of the flagship path set for each theme.

Return type *GnbrProcessor*

Returns A GnbrProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

indra.sources.gnbr.api.**process_chemical_disease_from_web**(*indicator_only=True*)
Call process_chemical_disease function on the GNBR datasets.

Parameters **indicator_only** (bool) – A switch to filter the data which is part of the flagship path set for each theme.

Return type *GnbrProcessor*

Returns A GnbrProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

indra.sources.gnbr.api.**process_chemical_gene**(*part1_path*, *part2_path*, *indicator_only=True*)
Process chemical–gene interactions.

Parameters

- **part1_path** (str) – Path to the first dataset of dependency paths and themes.
- **part2_path** (str) – Path to the second dataset of dependency paths and entity pairs.
- **indicator_only** (bool) – A switch to filter the data which is part of the flagship path set for each theme.

Return type *GnbrProcessor*

Returns A GnbrProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

indra.sources.gnbr.api.**process_chemical_gene_from_web**(*indicator_only=True*)
Call process_chemical_gene function on the GNBR datasets.

Parameters **indicator_only** (bool) – A switch to filter the data which is part of the flagship path set for each theme.

Return type *GnbrProcessor*

Returns A GnbrProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

indra.sources.gnbr.api.**process_from_files**(*part1_path*, *part2_path*, *first_type*, *second_type*, *indicator_only=True*)

Loading the databases from the given files.

Parameters

- **part1_path** (str) – Path to the first dataset which contains dependency paths and themes.
- **part2_path** (str) – Path to the second dataset which contains dependency paths and themes.
- **first_type** (str) – Type of the first agent.
- **second_type** (str) – Type of the second agent.
- **indicator_only** (bool) – A switch to filter the data which is part of the flagship path set for each theme.

Return type *GnbrProcessor*

Returns A GnbrProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

indra.sources.gnbr.api.**process_from_web**(*first_type*, *second_type*, *indicator_only=True*)

Loading the databases from the given urls.

Parameters

- **first_type** – Type of the first agent.
- **second_type** – Type of the second agent.
- **indicator_only** (bool) – A switch to filter the data which is part of the flagship path set for each theme.

Return type *GnbrProcessor*

Returns A GnbrProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

indra.sources.gnbr.api.**process_gene_disease**(*part1_path*, *part2_path*, *indicator_only=True*)

Process gene–disease interactions.

Parameters

- **part1_path** (str) – Path to the first dataset which contains dependency paths and themes.
- **part2_path** (str) – Path to the second dataset which contains dependency paths and entity pairs.
- **indicator_only** (bool) – A switch to filter the data which is part of the flagship path set for each theme.

Return type *GnbrProcessor*

Returns A GnbrProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

indra.sources.gnbr.api.**process_gene_disease_from_web**(*indicator_only=True*)

Call process_gene_disease function on the GNBR datasets.

Parameters **indicator_only** (bool) – A switch to filter the data which is part of the flagship path set for each theme.

Return type *GnbrProcessor*

Returns A GnbrProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

indra.sources.gnbr.api.**process_gene_gene**(*part1_path*, *part2_path*, *indicator_only=True*)
Process gene–gene interactions.

Parameters

- **part1_path** (str) – Path to the first dataset which contains dependency paths and themes.
- **part2_path** (str) – Path to the second dataset which contains dependency paths and entity pairs.
- **indicator_only** (bool) – A switch to filter the data which is part of the flagship path set for each theme.

Return type *GnbrProcessor*

Returns A GnbrProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

indra.sources.gnbr.api.**process_gene_gene_from_web**(*indicator_only=True*)
Call process_gene_gene function on the GNBR datasets.

Parameters **indicator_only** (bool) – A switch to filter the data which is part of the flagship path set for each theme.

Return type *GnbrProcessor*

Returns A GnbrProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

GNBR Processor (indra.sources.gnbr.processor)

This module contains the processor for GNBR. There are several, each corresponding to different kinds of interactions.

class indra.sources.gnbr.processor.**GnbrProcessor**(*df1*, *df2*, *first_type*, *second_type*, *indicator_only=True*)
A processor for interactions in the GNBR dataset.

Parameters

- **df1** (DataFrame) – Dataframe of dependency paths and themes.
- **df2** (DataFrame) – Dataframe of dependency paths and agents.
- **first_type** (str) – The type of the first entity in the data frame.
- **second_type** (str) – The type of the second entity in the data frame.

extract_stmts()
Extend the statements list with mappings.

indra.sources.gnbr.processor.**get_evidence**(*row*)
Return evidence for a Statement.

Parameters **row** (Series) – Currently investigated row of the dataframe.

Return type *Evidence*

Returns Evidence object with the source_api, the PMID and the original sentence.

indra.sources.gnbr.processor.**get_std_chemical**(*raw_string*, *db_id*)
Standardize chemical names.

Parameters

- **raw_string** (str) – Name of the agent in the GNBR dataset.
- **db_id** (str) – Entrez identifier of the agent.

Return type List[*Agent*]

Returns A standardized Agent object.

indra.sources.gnbr.processor.**get_std_disease**(*raw_string*, *db_id*)
Standardize disease names.

Parameters

- **raw_string** (str) – Name of the agent in the GNBR dataset.
- **db_id** (str) – Entrez identifier of the agent.

Return type List[*Agent*]

Returns A standardized Agent object.

indra.sources.gnbr.processor.**get_std_gene**(*raw_string*, *db_id*)
Standardize gene names.

Parameters

- **raw_string** (str) – Name of the agent in the GNBR dataset.
- **db_id** (str) – Entrez identifier of the agent.

Return type List[*Agent*]

Returns A standardized Agent object.

4.2.2 Molecular Pathway Databases

BEL (indra.sources.bel)

BEL API (indra.sources.bel.api)

High level API functions for the PyBEL processor.

indra.sources.bel.api.**process_bel_stmt**(*bel*, *squeeze=False*)
Process a single BEL statement and return the PybelProcessor or a single statement if `squeeze` is True.

Parameters

- **bel** (str) – A BEL statement. See example below.
- **squeeze** (Optional[bool]) – If squeeze and there's only one statement in the processor, it will be unpacked.

Returns **statements** – A list of INDRA statments derived from the BEL statement. If squeeze is true and there was only one statement, the unpacked INDRA statement will be returned.

Return type Union[*Statement*, *PybelProcessor*]

Examples

```
>>> from indra.sources.bel import process_bel_stmt
>>> bel_s = 'kin(p(FPLX:MEK)) -> kin(p(FPLX:ERK))'
>>> process_bel_stmt(bel_s, squeeze=True)
Activation(MEK(kinase), ERK(), kinase)
```

indra.sources.bel.api.**process_belscript**(*file_name*, ***kwargs*)
Return a PybelProcessor by processing a BEL script file.

Key word arguments are passed directly to pybel.from_path, for further information, see pybel.readthedocs.io/en/latest/io.html#pybel.from_path Some keyword arguments we use here differ from the defaults of PyBEL, namely we set *citation_clearing* to False and *no_identifier_validation* to True.

Parameters **file_name** (*str*) – The path to a BEL script file.

Returns **bp** – A PybelProcessor object which contains INDRA Statements in bp.statements.

Return type *PybelProcessor*

indra.sources.bel.api.**process_cbn_jgif_file**(*file_name*)
Return a PybelProcessor by processing a CBN JGIF JSON file.

Parameters **file_name** (*str*) – The path to a CBN JGIF JSON file.

Returns **bp** – A PybelProcessor object which contains INDRA Statements in bp.statements.

Return type *PybelProcessor*

indra.sources.bel.api.**process_json_file**(*file_name*)
Return a PybelProcessor by processing a Node-Link JSON file.

For more information on this format, see: http://pybel.readthedocs.io/en/latest/io.html#node-link-json

Parameters **file_name** (*str*) – The path to a Node-Link JSON file.

Returns **bp** – A PybelProcessor object which contains INDRA Statements in bp.statements.

Return type *PybelProcessor*

indra.sources.bel.api.**process_large_corpus**()
Return PybelProcessor with statements from Selventa Large Corpus.

Returns **bp** – A PybelProcessor object which contains INDRA Statements in its statements attribute.

Return type *PybelProcessor*

indra.sources.bel.api.**process_pybel_graph**(*graph*)
Return a PybelProcessor by processing a PyBEL graph.

Parameters **graph** (*pybel.struct.BELGraph*) – A PyBEL graph to process

Returns **bp** – A PybelProcessor object which contains INDRA Statements in bp.statements.

Return type *PybelProcessor*

indra.sources.bel.api.**process_pybel_neighborhood**(*entity_names*, *network_type='graph_jsongz_url'*, *network_file=None*, ***kwargs*)
Return PybelProcessor around neighborhood of given genes in a network.

This function processes the given network file and filters the returned Statements to ones that contain genes in the given list.

Parameters

- **entity_names** (*list[str]*) – A list of entity names (e.g., gene names) which will be used as the basis of filtering the result. If any of the Agents of an extracted INDRA Statement has a name appearing in this list, the Statement is retained in the result.
- **network_type** (*Optional[str]*) – The type of network that network_file is. The options are: belscript, json, cbn_jgif, graph_pickle, and graph_jsongz_url. Default: graph_jsongz_url
- **network_file** (*Optional[str]*) – Path to the network file/URL to process. If not given, by default, the Selventa Large Corpus is used via a URL pointing to a gzipped PyBEL Graph JSON file.

Returns **bp** – A PybelProcessor object which contains INDRA Statements in bp.statements.

Return type *PybelProcessor*

indra.sources.bel.api.**process_pybel_network**(*network_type*, *network_file*, ***kwargs*)
Return PybelProcessor by processing a given network file.

Parameters

- **network_type** (*str*) – The type of network that network_file is. The options are: belscript, json, cbn_jgif, graph_pickle, and graph_jsongz_url. Default: graph_jsongz_url
- **network_file** (*str*) – Path to the network file/URL to process.

Returns **bp** – A PybelProcessor object which contains INDRA Statements in bp.statements.

Return type *PybelProcessor*

indra.sources.bel.api.**process_small_corpus**()
Return PybelProcessor with statements from Selventa Small Corpus.

Returns **bp** – A PybelProcessor object which contains INDRA Statements in its statements attribute.

Return type *PybelProcessor*

PyBEL Processor (indra.sources.bel.processor)

Processor for PyBEL.

class indra.sources.bel.processor.**PybelProcessor**(*graph*)
Extract INDRA Statements from a PyBEL Graph.

Currently does not handle non-causal relationships (positiveCorrelation, (negativeCorrelation, hasVariant, etc.)

Parameters **graph** (*pybel.BELGraph*) – PyBEL graph containing the BEL content.

statements
A list of extracted INDRA Statements representing BEL Statements.

Type list[indra.statements.Statement]

indra.sources.bel.processor.**get_agent**(*node_data*, *node_modifier_data=None*)
Get an INDRA agent from a PyBEL node.

BioPAX (indra.sources.biopax)

This module allows processing BioPAX content into INDRA Statements. It uses the pybiopax package (https://github.com/indralab/pybiopax) to process OWL files or strings, or to obtain BioPAX content by querying the PathwayCommons web service. The module has been tested with BioPAX content from PathwayCommons https://www.pathwaycommons.org/archives/PC2/v12/. BioPAX from other sources may not adhere to the same conventions and could result in processing issues, though these can typically be addressed with minor changes in the processor's logic.

BioPAX API (indra.sources.biopax.api)

indra.sources.biopax.api.**process_model**(*model*)
Returns a BiopaxProcessor for a BioPAX model object.

Parameters **model** (*org.biopax.paxtools.model.Model*) – A BioPAX model object.

Returns **bp** – A BiopaxProcessor containing the obtained BioPAX model in bp.model.

Return type *BiopaxProcessor*

indra.sources.biopax.api.**process_owl**(*owl_filename*, *encoding=None*)
Returns a BiopaxProcessor for a BioPAX OWL file.

Parameters

- **owl_filename** (*str*) – The name of the OWL file to process.
- **encoding** (*Optional[str]*) – The encoding type to be passed to `pybiopax.model_from_owl_file()`.

Returns **bp** – A BiopaxProcessor containing the obtained BioPAX model in bp.model.

Return type *BiopaxProcessor*

indra.sources.biopax.api.**process_owl_str**(*owl_str*)
Returns a BiopaxProcessor for a BioPAX OWL file.

Parameters **owl_str** (*str*) – The string content of an OWL file to process.

Returns **bp** – A BiopaxProcessor containing the obtained BioPAX model in bp.model.

Return type *BiopaxProcessor*

indra.sources.biopax.api.**process_pc_neighborhood**(*gene_names*, *neighbor_limit=1*, *database_filter=None*)
Returns a BiopaxProcessor for a PathwayCommons neighborhood query.

The neighborhood query finds the neighborhood around a set of source genes.

http://www.pathwaycommons.org/pc2/#graph

http://www.pathwaycommons.org/pc2/#graph_kind

Parameters

- **gene_names** (*list*) – A list of HGNC gene symbols to search the neighborhood of. Examples: ['BRAF'], ['BRAF', 'MAP2K1']
- **neighbor_limit** (*Optional[int]*) – The number of steps to limit the size of the neighborhood around the gene names being queried. Default: 1

- **database_filter** (*Optional[list]*) – A list of database identifiers to which the query is restricted. Examples: ['reactome'], ['biogrid', 'pid', 'psp'] If not given, all databases are used in the query. For a full list of databases see http://www.pathwaycommons.org/pc2/datasources

Returns A BiopaxProcessor containing the obtained BioPAX model in its model attribute and a list of extracted INDRA Statements from the model in its statements attribute.

Return type *BiopaxProcessor*

indra.sources.biopax.api.**process_pc_pathsbetween**(*gene_names*, *neighbor_limit=1*, *database_filter=None*, *block_size=None*)

Returns a BiopaxProcessor for a PathwayCommons paths-between query.

The paths-between query finds the paths between a set of genes. Here source gene names are given in a single list and all directions of paths between these genes are considered.

http://www.pathwaycommons.org/pc2/#graph

http://www.pathwaycommons.org/pc2/#graph_kind

Parameters

- **gene_names** (*list*) – A list of HGNC gene symbols to search for paths between. Examples: ['BRAF', 'MAP2K1']
- **neighbor_limit** (*Optional[int]*) – The number of steps to limit the length of the paths between the gene names being queried. Default: 1
- **database_filter** (*Optional[list]*) – A list of database identifiers to which the query is restricted. Examples: ['reactome'], ['biogrid', 'pid', 'psp'] If not given, all databases are used in the query. For a full list of databases see http://www.pathwaycommons.org/pc2/datasources
- **block_size** (*Optional[int]*) – Large paths-between queries (above ~60 genes) can error on the server side. In this case, the query can be replaced by a series of smaller paths-between and paths-from-to queries each of which contains block_size genes.

Returns **bp** – A BiopaxProcessor containing the obtained BioPAX model in bp.model.

Return type *BiopaxProcessor*

indra.sources.biopax.api.**process_pc_pathsfromto**(*source_genes*, *target_genes*, *neighbor_limit=1*, *database_filter=None*)

Returns a BiopaxProcessor for a PathwayCommons paths-from-to query.

The paths-from-to query finds the paths from a set of source genes to a set of target genes.

http://www.pathwaycommons.org/pc2/#graph

http://www.pathwaycommons.org/pc2/#graph_kind

Parameters

- **source_genes** (*list*) – A list of HGNC gene symbols that are the sources of paths being searched for. Examples: ['BRAF', 'RAF1', 'ARAF']
- **target_genes** (*list*) – A list of HGNC gene symbols that are the targets of paths being searched for. Examples: ['MAP2K1', 'MAP2K2']
- **neighbor_limit** (*Optional[int]*) – The number of steps to limit the length of the paths between the source genes and target genes being queried. Default: 1
- **database_filter** (*Optional[list]*) – A list of database identifiers to which the query is restricted. Examples: ['reactome'], ['biogrid', 'pid', 'psp'] If not given, all databases are

used in the query. For a full list of databases see http://www.pathwaycommons.org/pc2/datasources

Returns **bp** – A BiopaxProcessor containing the obtained BioPAX model in bp.model.

Return type *BiopaxProcessor*

BioPAX Processor (indra.sources.biopax.processor)

class indra.sources.biopax.processor.**BiopaxProcessor**(*model*, *use_conversion_level_evidence=False*)

The BiopaxProcessor extracts INDRA Statements from a BioPAX model.

The BiopaxProcessor uses pattern searches in a BioPAX OWL model to extract mechanisms from which it constructs INDRA Statements.

Parameters **model** (*org.biopax.paxtools.model.Model*) – A BioPAX model object (java object)

model

A BioPAX model object (java object) which is queried using Paxtools to extract INDRA Statements

Type org.biopax.paxtools.model.Model

statements

A list of INDRA Statements that were extracted from the model.

Type list[indra.statements.Statement]

eliminate_exact_duplicates()

Eliminate Statements that were extracted multiple times.

Due to the way the patterns are implemented, they can sometimes yield the same Statement information multiple times, in which case, we end up with redundant Statements that aren't from independent underlying entries. To avoid this, here, we filter out such duplicates.

feature_delta(*from_pe*, *to_pe*)

Return gained and lost modifications and any activity change.

static find_matching_entities(*left_simple*, *right_simple*)

Find matching entities between two lists of simple entities.

static find_matching_left_right(*conversion*)

Find matching entities on the left and right of a conversion.

get_activity_modification()

Extract INDRA ActiveForm statements from the BioPAX model.

get_conversions()

Extract Conversion INDRA Statements from the BioPAX model.

get_gap_gef()

Extract Gap and Gef INDRA Statements.

get_modifications()

Extract INDRA Modification Statements from the BioPAX model.

get_regulate_activities()

Get Activation/Inhibition INDRA Statements from the BioPAX model.

get_regulate_amounts()

Extract INDRA RegulateAmount Statements from the BioPAX model.

static mod_condition_from_mod_feature(*mf*)
Extract the type of modification and the position from a ModificationFeature object in the INDRA format.

save_model(*file_name*)
Save the BioPAX model object in an OWL file.

Parameters **file_name** (*str*) – The name of the OWL file to save the model in.

SIGNOR (indra.sources.signor)

SIGNOR API (indra.sources.signor.api)

indra.sources.signor.api.**process_from_file**(*signor_data_file*, *signor_complexes_file=None*)
Process Signor interaction data from CSV files.

Parameters

- **signor_data_file** (*str*) – Path to the Signor interaction data file in CSV format.
- **signor_complexes_file** (*str*) – Path to the Signor complexes data in CSV format. If unspecified, Signor complexes will not be expanded to their constitutents.

Returns SignorProcessor containing Statements extracted from the Signor data.

Return type indra.sources.signor.SignorProcessor

SIGNOR Processor (indra.sources.signor.processor)

An input processor for the SIGNOR database: a database of causal relationships between biological entities.

See publication:

Perfetto et al., "SIGNOR: a database of causal relationships between biological entities," Nucleic Acids Research, Volume 44, Issue D1, 4 January 2016, Pages D548-D554. https://doi.org/10.1093/nar/gkv1048

class indra.sources.signor.processor.**SignorProcessor**(*data*, *complex_map=None*)
Processor for Signor dataset, available at http://signor.uniroma2.it.

Parameters

- **data** (*iterator*) – Iterator over rows of a SIGNOR CSV file.
- **complex_map** (*dict*) – A dict containing SIGNOR complexes, keyed by their IDs.

statements
A list of INDRA Statements extracted from the SIGNOR table.

Type list[indra.statements.Statements]

no_mech_rows
List of rows where no mechanism statements were generated.

Type list of SignorRow namedtuples

no_mech_ctr
Counter listing the frequency of different MECHANISM types in the list of no-mechanism rows.

Type collections.Counter

BioGrid (indra.sources.biogrid)

class indra.sources.biogrid.**BiogridProcessor**(*biogrid_file=None*, *physical_only=True*)
Extracts INDRA Complex statements from Biogrid interaction data.

Parameters

- **biogrid_file** (*str*) – The file containing the Biogrid data in .tab2 format. If not provided, the BioGrid data is downloaded from the BioGrid website.
- **physical_only** (*boolean*) – If True, only physical interactions are included (e.g., genetic interactions are excluded). If False, all interactions are included).

statements
Extracted INDRA Complex statements.

Type list[indra.statements.Statements]

physical_only
Indicates whether only physical interactions were included during statement processing.

Type boolean

Human Protein Reference Database (indra.sources.hprd)

This module implements getting content from the Human Protein Reference Database (HPRD), a curated protein data resource, as INDRA Statements. In particular, the module supports extracting post-translational modifications, protein complexes, and (binary) protein-protein interactions from HPRD.

More information about HPRD can be obtained at http://www.hprd.org and in these publications:

- Peri, S. et al. (2003). Development of Human Protein Reference Database as an initial platform for approaching systems biology in humans. Genome Research. 13, 2363-2371.
- Prasad, T. S. K. et al. (2009). Human Protein Reference Database - 2009 Update. Nucleic Acids Research. 37, D767-72.

Data from the final release of HPRD (version 9) can be obtained at the following URLs:

- http://www.hprd.org/RELEASE9/HPRD_FLAT_FILES_041310.tar.gz (text files)
- http://www.hprd.org/RELEASE9/HPRD_XML_041310.tar.gz (XML)

This module is designed to process the text files obtained from the first link listed above.

HPRD API (indra.sources.hprd.api)

indra.sources.hprd.api.**process_flat_files**(*id_mappings_file*, *complexes_file=None*, *ptm_file=None*, *ppi_file=None*, *seq_file=None*, *motif_window=7*)
Get INDRA Statements from HPRD data.

Of the arguments, *id_mappings_file* is required, and at least one of *complexes_file*, *ptm_file*, and *ppi_file* must also be given. If *ptm_file* is given, *seq_file* must also be given.

Note that many proteins (> 1,600) in the HPRD content are associated with outdated RefSeq IDs that cannot be mapped to Uniprot IDs. For these, the Uniprot ID obtained from the HGNC ID (itself obtained from the Entrez ID) is used. Because the sequence referenced by the Uniprot ID obtained this way may be different from the (outdated) RefSeq sequence included with the HPRD content, it is possible that this will lead to invalid site positions with respect to the Uniprot IDs.

To allow these site positions to be mapped during assembly, the Modification statements produced by the HprdProcessor include an additional key in the *annotations* field of their Evidence object. The annotations field is called 'site_motif' and it maps to a dictionary with three elements: 'motif', 'respos', and 'off_by_one'. 'motif' gives the peptide sequence obtained from the RefSeq sequence included with HPRD. 'respos' indicates the position in the peptide sequence containing the residue. Note that these positions are ONE-INDEXED (not zero-indexed). Finally, the 'off-by-one' field contains a boolean value indicating whether the correct position was inferred as being an off-by-one (methionine cleavage) error. If True, it means that the given residue could not be found in the HPRD RefSeq sequence at the given position, but a matching residue was found at position+1, suggesting a sequence numbering based on the methionine-cleaved sequence. The peptide included in the 'site_motif' dictionary is based on this updated position.

Parameters

- **id_mappings_file** (*str*) – Path to HPRD_ID_MAPPINGS.txt file.
- **complexes_file** (*Optional[str]*) – Path to PROTEIN_COMPLEXES.txt file.
- **ptm_file** (*Optional[str]*) – Path to POST_TRANSLATIONAL_MODIFICATIONS.txt file.
- **ppi_file** (*Optional[str]*) – Path to BINARY_PROTEIN_PROTEIN_INTERACTIONS.txt file.
- **seq_file** (*Optional[str]*) – Path to PROTEIN_SEQUENCES.txt file.
- **motif_window** (*int*) – Number of flanking amino acids to include on each side of the PTM target residue in the 'site_motif' annotations field of the Evidence for Modification Statements. Default is 7.

Returns An HprdProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

Return type *HprdProcessor*

HPRD Processor (indra.sources.hprd.processor)

class indra.sources.hprd.processor.**HprdProcessor**(*id_df*, *cplx_df=None*, *ptm_df=None*, *ppi_df=None*, *seq_dict=None*, *motif_window=7*)

Get INDRA Statements from HPRD data.

See documentation for *indra.sources.hprd.api.process_flat_files*.

Parameters

- **id_df** (*pandas.DataFrame*) – DataFrame loaded from the HPRD_ID_MAPPINGS.txt file.
- **cplx_df** (*pandas.DataFrame*) – DataFrame loaded from the PROTEIN_COMPLEXES.txt file.
- **ptm_df** (*pandas.DataFrame*) – DataFrame loaded from the POST_TRANSLATIONAL_MODIFICATIONS.txt file.
- **ppi_df** (*pandas.DataFrame*) – DataFrame loaded from the BINARY_PROTEIN_PROTEIN_INTERACTIONS.txt file.
- **seq_dict** (*dict*) – Dictionary mapping RefSeq IDs to protein sequences, loaded from the PROTEIN_SEQUENCES.txt file.
- **motif_window** (*int*) – Number of flanking amino acids to include on each side of the PTM target residue in the 'site_motif' annotations field of the Evidence for Modification Statements. Default is 7.

statements
INDRA Statements (Modifications and Complexes) produced from the HPRD content.

Type list of INDRA Statements

id_df
DataFrame loaded from HPRD_ID_MAPPINGS.txt file.

Type pandas.DataFrame

seq_dict
Dictionary mapping RefSeq IDs to protein sequences, loaded from the PROTEIN_SEQUENCES.txt file.

no_hgnc_for_egid
Counter listing Entrez gene IDs reference in the HPRD content that could not be mapped to a current HGNC ID, along with their frequency.

Type collections.Counter

no_up_for_hgnc
Counter with tuples of form (entrez_id, hgnc_symbol, hgnc_id) where the HGNC ID could not be mapped to a Uniprot ID, along with their frequency.

Type collections.Counter

no_up_for_refseq
Counter of RefSeq protein IDs that could not be mapped to any Uniprot ID, along with frequency.

Type collections.Counter

many_ups_for_refseq
Counter of RefSeq protein IDs that yielded more than one matching Uniprot ID. Note that in these cases, the Uniprot ID obtained from HGNC is used.

Type collections.Counter

invalid_site_pos
List of tuples of form (refseq_id, residue, position) indicating sites of post translational modifications where the protein sequences provided by HPRD did not contain the given residue at the given position.

Type list of tuples

off_by_one
The subset of sites contained in *invalid_site_pos* where the given residue can be found at position+1 in the HPRD protein sequence, suggesting an off-by-one error due to numbering based on the protein with initial methionine cleaved. Note that no mapping is performed by the processor.

Type list of tuples

motif_window
Number of flanking amino acids to include on each side of the PTM target residue in the 'site_motif' annotations field of the Evidence for Modification Statements. Default is 7.

Type int

get_complexes(*cplx_df*)
Generate Complex Statements from the HPRD protein complexes data.

Parameters **cplx_df** (*pandas.DataFrame*) – DataFrame loaded from the PROTEIN_COMPLEXES.txt file.

get_ppis(*ppi_df*)
Generate Complex Statements from the HPRD PPI data.

Parameters **ppi_df** (*pandas.DataFrame*) – DataFrame loaded from the BINARY_PROTEIN_PROTEIN_INTERACTIONS.txt file.

get_ptms(*ptm_df*)
Generate Modification statements from the HPRD PTM data.

Parameters **ptm_df** (*pandas.DataFrame*) – DataFrame loaded from the POST_TRANSLATIONAL_MODIFICATIONS.txt file.

TRRUST Database (indra.sources.trrust)

This module provides an interface to the TRRUST knowledge base and extracts TF-target relationships as INDRA Statements.

TRRUST is available at https://www.grnpedia.org/trrust/, see also https://www.ncbi.nlm.nih.gov/pubmed/29087512.

TRRUST API (indra.sources.trrust.api)

indra.sources.trrust.api.**process_from_web**()
Return a TrrustProcessor based on the online interaction table.

Returns A TrrustProcessor object that has a list of INDRA Statements in its statements attribute.

Return type *TrrustProcessor*

TRRUST Processor (indra.sources.trrust.processor)

class indra.sources.trrust.processor.**TrrustProcessor**(*df*)
Processor to extract INDRA Statements from Trrust data frame.

df
The Trrust table to process.

Type pandas.DataFrame

statements
The list of INDRA Statements extracted from the table.

Type list[indra.statements.Statement]

extract_statements()
Process the table to extract Statements.

indra.sources.trrust.processor.**get_grounded_agent**(*gene_name*)
Return a grounded Agent based on an HGNC symbol.

indra.sources.trrust.processor.**make_stmt**(*stmt_cls*, *tf_agent*, *target_agent*, *pmid*)
Return a Statement based on its type, agents, and PMID.

Phospho.ELM (indra.sources.phosphoelm)

This module provides an interface to the Phospho.ELM database and extracts phosphorylation relationships as INDRA Statements. Phospho.ELM is available at http://phospho.elm.eu.org/, see also https://academic.oup.com/nar/article/39/suppl_1/D261/2506728

Phospho.ELM API (indra.sources.phosphoelm.api)

indra.sources.phosphoelm.api.**process_from_dump**(*fname*, *delimiter='\t'*)
Process a Phospho.ELM file dump

The dump can be obtained at http://phospho.elm.eu.org/dataset.html.

Parameters

- **fname** (*str*) – File path to the phospho.ELM file dump.
- **delimiter** (*str*) – The delimiter to use for csv.reader

Returns An instance of a PhosphoElmProcessor containing the statements generated from the file dump

Return type indra.sources.phosphoelm.PhosphoElmProcessor

Phospho.ELM Processor (indra.sources.phosphoelm.processor)

class indra.sources.phosphoelm.processor.**PhosphoElmProcessor**(*phosphoelm_data*)
Processes data dumps from the phospho.ELM database.

See http://phospho.elm.eu.org/dataset.html

Parameters **phosphoelm_data** (*list[dict]*) – JSON compatible list of entries from a phospho.ELM data dump

statements
A list of the phosphorylation statements produced by the entries in phosphoelm_data

Type list[indra.statements.Phosphorylation]

process_phosphorylations(*skip_empty=True*)
Create Phosphorylation statements from phosphoelm_data

Parameters **skip_empty** (*bool*) – Default: True. If False, also create statements when upstream kinases in entry['kinases'] are not known.

VirHostNet (indra.sources.virhostnet)

This module implements an API for VirHostNet 2.0 (http://virhostnet.prabi.fr/).

VirHostNet API (indra.sources.virhostnet.api)

indra.sources.virhostnet.api.**process_df**(*df*, *up_web_fallback=False*)
Process a VirHostNet pandas DataFrame.

Parameters **df** (*pandas.DataFrame*) – A DataFrame representing VirHostNet interactions (in the same format as the web service).

Returns A VirhostnetProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

Return type *VirhostnetProcessor*

indra.sources.virhostnet.api.**process_from_web**(*query=None*, *up_web_fallback=False*)
Process host-virus interactions from the VirHostNet website.

Parameters **query** (*Optional[str]*) – A query that constrains the results to a given subset of the VirHostNet database. Example: "taxid:2697049" to search for interactions for SARS-CoV-2. If not provided, By default, the "*" query is used which returns the full database.

Returns A VirhostnetProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

Return type *VirhostnetProcessor*

indra.sources.virhostnet.api.**process_tsv**(*fname*, *up_web_fallback=False*)
Process a TSV data file obtained from VirHostNet.

Parameters **fname** (*str*) – The path to the VirHostNet tabular data file (in the same format as the web service).

Returns A VirhostnetProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

Return type *VirhostnetProcessor*

VirHostNet Processor (indra.sources.virhostnet.processor)

class indra.sources.virhostnet.processor.**VirhostnetProcessor**(*df*, *up_web_fallback=False*)
A processor that takes a pandas DataFrame and extracts INDRA Statements.

Parameters **df** (*pandas.DataFrame*) – A pandas DataFrame representing VirHostNet interactions.

df
A pandas DataFrame representing VirHostNet interactions.

Type pandas.DataFrame

statements
A list of INDRA Statements extracted from the DataFrame.

Type list[indra.statements.Statement]

indra.sources.virhostnet.processor.**get_agent_from_grounding**(*grounding*, *up_web_fallback=False*)
Return an INDRA Agent based on a grounding annotation.

indra.sources.virhostnet.processor.**parse_psi_mi**(*psi_mi_str*)
Parse a PSI-MI annotation into an ID and name pair.

indra.sources.virhostnet.processor.**parse_source_ids**(*source_id_str*)
Parse VirHostNet source id annotations into a dict.

indra.sources.virhostnet.processor.**parse_text_refs**(*text_ref_str*)
Parse a text reference annotation into a text_refs dict.

indra.sources.virhostnet.processor.**process_row**(*row*, *up_web_fallback=False*)
Process one row of the DataFrame into an INDRA Statement.

OmniPath (indra.sources.omnipath)

The OmniPath module accesses biomolecular interaction data from various curated databases using the OmniPath API (see https://saezlab.github.io/pypath/html/index.html#webservice) and processes the returned data into statements using the OmniPathProcessor.

Currently, the following data is collected:

- Modifications from the PTMS endpoint https://saezlab.github.io/pypath/html/index.html#enzyme-substrate-interactions
- Ligand-Receptor data from the interactions endpoint https://saezlab.github.io/pypath/html/index.html#interaction-datasets

To process all statements, use the function *process_from_web*:

```
>>> from indra.sources.omnipath import process_from_web
>>> omnipath_processor = process_from_web()
>>> stmts = omnipath_processor.statements
```

OmniPath API (indra.sources.omnipath.api)

indra.sources.omnipath.api.**process_from_web**()
Query the OmniPath web API and return an OmniPathProcessor.

Returns An OmniPathProcessor object which contains a list of extracted INDRA Statements in its statements attribute.

Return type *OmniPathProcessor*

OmniPath Processor (indra.sources.omnipath.processor)

class indra.sources.omnipath.processor.**OmniPathProcessor**(*ptm_json=None*, *ligrec_json=None*)
Class to process OmniPath JSON into INDRA Statements.

process_ligrec_interactions()
Process ligand-receptor json if present

process_ptm_mods()
Process ptm json if present

UbiBrowser Database (indra.sources.ubibrowser)

This module implements an API and processor for UbiBrowser, a resource for ubiquitin ligase-sbstrate and deubiquitinase-substrate interactions. See http://ubibrowser.ncpsb.org.cn/ for more information.

UbiBrowser API (indra.sources.ubibrowser.api)

indra.sources.ubibrowser.api.**process_df**(*e3_df*, *dub_df*)
Process data frames containing UbiBrowser data.

Parameters

- **e3_df** (DataFrame) – A data frame containing UbiBrowser E3 data.
- **dub_df** (DataFrame) – A data frame containing UbiBrowser DUB data.

Return type *UbiBrowserProcessor*

Returns An UbiBrowserProcessor object with INDRA Statements extracted in its statements attribute.

indra.sources.ubibrowser.api.**process_file**(*e3_path*, *dub_path*)
Process UbiBrowser data from files.

Parameters

- **e3_path** (str) – The path to the E3 file.
- **dub_path** (str) – The path to the DUB file.

Return type *UbiBrowserProcessor*

Returns An UbiBrowserProcessor object with INDRA Statements extracted in its statements attribute.

indra.sources.ubibrowser.api.**process_from_web**()
Download the UbiBrowser data from the web and process it.

Return type *UbiBrowserProcessor*

Returns An UbiBrowserProcessor object with INDRA Statements extracted in its statements attribute.

UbiBrowser Processor (indra.sources.ubibrowser.processor)

class indra.sources.ubibrowser.processor.**UbiBrowserProcessor**(*e3_df*, *dub_df*)
Processor for UbiBrowser data.

ACSN Database (indra.sources.acsn)

This module implements an API and processor for the ACSN resource which is available at https://acsn.curie.fr/ACSN2/ACSN2.html.

ACSN API (indra.sources.acsn.api)

indra.sources.acsn.api.**process_df**(*relations_df*, *correspondence_dict*)
Process ACSN data from input data structures.

Parameters

- **relations_df** (DataFrame) – An ACSN tab-separated data frame which consists of binary relationships between proteins with PMIDs.
- **correspondence_dict** (Mapping) – A dictionary with correspondences between ACSN entities and their HGNC symbols.

Return type *AcsnProcessor*

Returns A processor with a list of INDRA statements that were extracted in its statements attribute.

indra.sources.acsn.api.**process_files**(*relations_path*, *correspondence_path*)
Process ACSN data from input files.

Parameters

- **relations_path** (str) – Path to the ACSN binary relations file.
- **correspondence_path** (str) – Path to the ACSN correspondence GMT file.

Return type *AcsnProcessor*

Returns A processor with a list of INDRA statements that were extracted in its statements attribute.

indra.sources.acsn.api.**process_from_web**()
Process ACSN data directly from the web.

Return type *AcsnProcessor*

Returns A processor with a list of INDRA statements that were extracted in its statements attribute.

ACSN Processor (indra.sources.acsn.processor)

class indra.sources.acsn.processor.**AcsnProcessor**(*relations_df*, *correspondence_dict*)
Processes Atlas of cancer signalling network (ACSN) relationships into INDRA statements

relations_df
A tab-separated data frame which consists of binary relationship between proteins with PMIDs.

Type pandas.DataFrame

correspondence_dict
A dictionary with correspondences between ACSN entities and their HGNC symbols.

Type dict

extract_statements()
Return INDRA Statements Extracted from ACSN relations.

get_agent(*acsn_agent*)
Return an INDRA Agent corresponding to an ACSN agent.

Parameters **acsn_agent** (str) – Agent extracted from the relations statement data frame

Return type Optional[*Agent*]

Returns Returns INDRA agent with HGNC or FamPlex ID in db_refs. If there are no groundings available, we return None.

indra.sources.acsn.processor.**get_stmt_type**(*stmt_type*)
Return INDRA statement type from ACSN relation.

Parameters **stmt_type** (str) – An ACSN relationship type

Return type Optional[*Statement*]

Returns INDRA equivalent of the ACSN relation type or None if a mappings is not available.

4.2.3 Chemical Information Databases

CTD (indra.sources.ctd)

This module implements an API and processor to extract INDRA Statements from the Comparative Toxicogenomics Database (CTD), see http://ctdbase.org/. It currently extracts chemical-gene, gene-disease, and chemical-disease relationships. In particular, it extracts the curated (not inferred) and directional/causal relationships from these subsets.

CTD API (indra.sources.ctd.api)

indra.sources.ctd.api.**process_dataframe**(*df*, *subset*)
Process a subset of CTD from a DataFrame into INDRA Statements.

Parameters

- **df** (*pandas.DataFrame*) – A DataFrame of the given CTD subset.
- **subset** (*str*) – A CTD subset, one of chemical_gene, chemical_disease, gene_disease.

Returns A CTDProcessor which contains INDRA Statements extracted from the given CTD subset as its statements attribute.

Return type *CTDProcessor*

indra.sources.ctd.api.**process_from_web**(*subset*, *url=None*)
Process a subset of CTD from the web into INDRA Statements.

Parameters

- **subset** (*str*) – A CTD subset, one of chemical_gene, chemical_disease, gene_disease.
- **url** (*Optional[str]*) – If not provided, the default CTD URL is used (beware, it usually gives permission denied). If provided, the given URL is used to access a tsv or tsv.gz file.

Returns A CTDProcessor which contains INDRA Statements extracted from the given CTD subset as its statements attribute.

Return type *CTDProcessor*

indra.sources.ctd.api.**process_tsv**(*fname*, *subset*)
Process a subset of CTD from a tsv or tsv.gz file into INDRA Statements.

Parameters

- **fname** (*str*) – Path to a tsv or tsv.gz file of the given CTD subset.
- **subset** (*str*) – A CTD subset, one of chemical_gene, chemical_disease, gene_disease.

Returns A CTDProcessor which contains INDRA Statements extracted from the given CTD subset as its statements attribute.

Return type *CTDProcessor*

CTD Processor (indra.sources.ctd.processor)

class indra.sources.ctd.processor.**CTDChemicalDiseaseProcessor**(*df*)
Processes chemical-disease relationships from CTD.

class indra.sources.ctd.processor.**CTDChemicalGeneProcessor**(*df*)
Processes chemical-gene relationships from CTD.

class indra.sources.ctd.processor.**CTDGeneDiseaseProcessor**(*df*)
Processes gene-disease relationships from CTD.

class indra.sources.ctd.processor.**CTDProcessor**(*df*)
Parent class for CTD relation-specific processors.

DrugBank (indra.sources.drugbank)

This module provides an API and processor for DrugBank content. It builds on the XML-formatted data schema of DrugBank and expects the XML file to be available locally. The full DrugBank download can be obtained at: https://www.drugbank.ca/releases/latest. Once the XML file is decompressed, it can be processed using the process_xml function.

Alternatively, the latest DrugBank data can be automatically loaded via `drugbank_downloader` with the following code after doing `pip install drugbank_downloader bioversions`:

```
import pickle
import indra.sources.drugbank
processor = indra.sources.drugbank.get_drugbank_processor()
with open('drugbank_indra_statements.pkl', 'wb') as file:
    pickle.dump(processor.statements, file, protocol=pickle.HIGHEST_PROTOCOL)
```

DrugBank API (indra.sources.drugbank.api)

indra.sources.drugbank.api.**process_element_tree**(*et*)
Return a processor by extracting Statement from DrugBank XML.

Parameters **et** (*xml.etree.ElementTree*) – An ElementTree loaded from the DrugBank XML file to process.

Returns A DrugbankProcessor instance which contains a list of INDRA Statements in its statements attribute that were extracted from the given ElementTree.

Return type *DrugbankProcessor*

indra.sources.drugbank.api.**process_from_web**(*username=None*, *password=None*, *version=None*, *prefix=None*)
Get a processor using *process_xml()* with `drugbank_downloader`.

Parameters

- **username** (Optional[str]) – The DrugBank username. If not passed, looks up in the environment `DRUGBANK_USERNAME`. If not found, raises a ValueError.
- **password** (Optional[str]) – The DrugBank password. If not passed, looks up in the environment `DRUGBANK_PASSWORD`. If not found, raises a ValueError.
- **version** (Optional[str]) – The DrugBank version. If not passed, uses `bioversions` to look up the most recent version.

- **prefix** (Union[None, str, Sequence[str]]) – The prefix and subkeys passed to `pystow.ensure()` to specify a non-default location to download the data to.

Returns A DrugbankProcessor instance which contains a list of INDRA Statements in its statements attribute that were extracted from the given DrugBank version

Return type *DrugbankProcessor*

indra.sources.drugbank.api.**process_xml**(*fname*)
Return a processor by extracting Statements from DrugBank XML.

Parameters **fname** (*str*) – The path to a DrugBank XML file to process.

Returns A DrugbankProcessor instance which contains a list of INDRA Statements in its statements attribute that were extracted from the given XML file.

Return type *DrugbankProcessor*

DrugBank Processor (indra.sources.drugbank.processor)

class indra.sources.drugbank.processor.**DrugbankProcessor**(*xml_tree*)
Processor to extract INDRA Statements from DrugBank content.

The processor assumes that an ElementTree is available which it then traverses to find drug-target information.

Parameters **xml_tree** (*xml.etree.ElementTree.ElementTree*) – An XML ElementTree representing DrugBank XML content.

statements
A list of INDRA Statements that were extracted from DrugBank content.

Type list of indra.statements.Statement

Drug Gene Interaction (DGI) Database (indra.sources.dgi)

A processor for the Drug Gene Interaction DB.

- Integration of the Drug–Gene Interaction Database (DGIdb 4.0) with open crowdsource efforts. Freshour, *et al*. Nucleic Acids Research. 2020 Nov 25.

Interactions data from the January 2021 release can be obtained at the following URLs:

- https://www.dgidb.org/data/monthly_tsvs/2021-Jan/interactions.tsv

DGI API (indra.sources.dgi.api)

API for Drug Gene Interaction DB.

indra.sources.dgi.api.**get_version_df**(*version=None*)
Get the latest version of the DGI interaction dataframe.

Return type Tuple[str, DataFrame]

indra.sources.dgi.api.**process_df**(*df*, *version=None*, *skip_databases=None*)
Get a processor that extracted INDRA Statements from DGI content based on the given dataframe.

Parameters

- **df** (*pd.DataFrame*) – A pandas DataFrame for the DGI interactions file.

- **version** (*Optional[str]*) – The optional version of DGI to use. If not given, statements will not be annotated with a version number.
- **skip_databases** (*Optional[set[str]]*) – A set of primary database sources to skip. If not given, DrugBank is skipped since there is a dedicated module in INDRA for obtaining DrugBank statements.

Returns **dp** – A DGI processor with pre-extracted INDRA statements

Return type *DGIProcessor*

indra.sources.dgi.api.**process_version**(*version=None*, *skip_databases=None*)
Get a processor that extracted INDRA Statements from DGI content.

Parameters

- **version** (*Optional[str]*) – The optional version of DGI to use. If not given, the version is automatically looked up.
- **skip_databases** (*Optional[set[str]]*) – A set of primary database sources to skip. If not given, DrugBank is skipped since there is a dedicated module in INDRA for obtaining DrugBank statements.

Returns **dp** – A DGI processor with pre-extracted INDRA statements

Return type *DGIProcessor*

DGI Processor (indra.sources.dgi.processor)

Processor for the Drug Gene Interaction DB.

class indra.sources.dgi.processor.**DGIProcessor**(*df=None*, *version=None*, *skip_databases=None*)
Processor to extract INDRA Statements from DGI content.

Parameters

- **df** (*pd.DataFrame*) – A pandas DataFrame for the DGI interactions file. If none given, the most recent version will be automatically looked up.
- **version** (*str*) – The optional version of DGI to use. If no `df` is given, this is also automatically looked up.

extract_statements()
Extract statements from DGI.

Return type List[*Statement*]

row_to_statements(*gene_name*, *ncbigene_id*, *source*, *interactions*, *drug_name*, *drug_curie*, *pmids*)
Convert a row in the DGI dataframe to a statement.

Return type Iterable[*Statement*]

statements: List[*indra.statements.statements.Statement*]
A list of INDRA Statements that were extracted from DGI content.

Target Affinity Spectrum (indra.sources.tas)

This module provides and API and processor to the Target Affinity Spectrum data set compiled by N. Moret in the Laboratory of Systems Pharmacology at HMS. This data set is based on experiments as opposed to the manually curated drug-target relationships provided in the LINCS small molecule dataset.

Moret, N., et al. (2018). Cheminformatics tools for analyzing and designing optimized small molecule libraries. BioRxiv, (617), 358978. https://doi.org/10.1101/358978

TAS API (indra.sources.tas.api)

indra.sources.tas.api.**process_csv**(*fname*, *affinity_class_limit=2*, *named_only=False*, *standardized_only=False*)

Return a TasProcessor for the contents of a given CSV file..

Interactions are classified into the following classes based on affinity:

1 – Kd < 100nM
2 – 100nM < Kd < 1uM
3 – 1uM < Kd < 10uM
10 – Kd > 10uM

By default, only classes 1 and 2 are extracted but the affinity_class_limit parameter can be used to change the upper limit of extracted classes.

Parameters

- **fname** (*str*) – The path to a local CSV file containing the TAS data.
- **affinity_class_limit** (*Optional[int]*) – Defines the highest class of binding affinity that is included in the extractions. Default: 2
- **named_only** (*Optional[bool]*) – If True, only chemicals that have a name assigned in some name space (including ones that aren't fully stanadardized per INDRA's ontology, e.g., CHEMBL1234) are included. If False, chemicals whose name is assigned based on an ID (e.g., CHEMBL)rather than an actual name are also included. Default: False
- **standardized_only** (*Optional[bool]*) – If True, only chemicals that are fully standardized per INDRA's ontology (i.e., they have grounding appearing in one of the default_ns_order name spaces, and consequently have any groundings and their name standardized) are extracted. Default: False

Returns A TasProcessor object which has a list of INDRA Statements extracted from the CSV file representing drug-target inhibitions in its statements attribute.

Return type *TasProcessor*

indra.sources.tas.api.**process_from_web**(*affinity_class_limit=2*, *named_only=False*, *standardized_only=False*)

Return a TasProcessor for the contents of the TAS dump online.

Interactions are classified into the following classes based on affinity:

1 – Kd < 100nM
2 – 100nM < Kd < 1uM
3 – 1uM < Kd < 10uM
10 – Kd > 10uM

By default, only classes 1 and 2 are extracted but the affinity_class_limit parameter can be used to change the upper limit of extracted classes.

Parameters

- **affinity_class_limit** (*Optional[int]*) – Defines the highest class of binding affinity that is included in the extractions. Default: 2
- **named_only** (*Optional[bool]*) – If True, only chemicals that have a name assigned in some name space (including ones that aren't fully stanadardized per INDRA's ontology, e.g., CHEMBL1234) are included. If False, chemicals whose name is assigned based on an ID (e.g., CHEMBL)rather than an actual name are also included. Default: False
- **standardized_only** (*Optional[bool]*) – If True, only chemicals that are fully standardized per INDRA's ontology (i.e., they have grounding appearing in one of the default_ns_order name spaces, and consequently have any groundings and their name standardized) are extracted. Default: False

Returns A TasProcessor object which has a list of INDRA Statements extracted from the CSV file representing drug-target inhibitions in its statements attribute.

Return type *TasProcessor*

TAS Processor (indra.sources.tas.processor)

class indra.sources.tas.processor.**TasProcessor**(*data*, *affinity_class_limit=2*, *named_only=False*, *standardized_only=False*)

A processor for the Target Affinity Spectrum data table.

CRoG (indra.sources.crog)

Processor for the Chemical Roles Graph (CRoG).

Contains axiomization of ChEBI roles, their targets, and actual relationship polarity.

- Extension of Roles in the ChEBI Ontology. Hoyt, C. T., *et al.* (2020). *ChemRxiv*, 12591221.

CRoG API (indra.sources.crog.api)

API for the Chemical Roles Graph (CRoG).

indra.sources.crog.api.**process_from_web**(*url=None*)

Process statements from CRoG over the web.

Parameters **url** (Optional[str]) – An optional URL. If none given, defaults to indra.sources.crog.processor.CROG_URL.

Returns **processor** – A processor with pre-extrated statements.

Return type *CrogProcessor*

CRoG Processor (indra.sources.crog.processor)

Processor for the Chemical Roles Graph (CRoG).

class indra.sources.crog.processor.**CrogProcessor**(*url=None*)
A processor for the Chemical Roles Graph.

Parameters **url** (*str*) – An optional URL. If none given, defaults to indra.sources.crog.processor.CROG_URL.

extract_statements()
Extract statements from the remote JSON file.

CREEDS (indra.sources.creeds)

Processor for the CRowd Extracted Expression of Differential Signatures (CREEDS) from [wang2016].

Contains results of differential gene expression experiments extracted from the Gene Expression Omnibus due to three types of perturbations:

1. Single gene knockdown/knockout
2. Drug
3. Disease

CREEDS API (indra.sources.creeds.api)

API for CREEDS.

indra.sources.creeds.api.**process_from_file**(*path*, *entity_type*)
Process statements from CREEDS in a file.

Parameters

- **path** (Union[str, Path]) – The path to a JSON file containing records for the CREEDS data
- **entity_type** (str) – Either ‘gene’, ‘disease’, or ‘chemical’ to specify which dataset to get.

Return type CREEDSProcessor

Returns A processor with pre-extracted statements.

indra.sources.creeds.api.**process_from_web**(*entity_type*)
Process statements from CREEDS by automatially downloading them.

Parameters **entity_type** (str) – Either ‘gene’, ‘disease’, or ‘chemical’ to specify which dataset to get.

Return type CREEDSProcessor

Returns A processor with pre-extracted statements.

CREEDS Processor (indra.sources.creeds.processor)

Processors for CREEDS data.

class indra.sources.creeds.processor.**CREEDSChemicalProcessor**(*records*)
A processor for chemical perturbation experiments in CREEDS.

class indra.sources.creeds.processor.**CREEDSDiseaseProcessor**(*records*)
A processor for disease perturbation experiments in CREEDS.

class indra.sources.creeds.processor.**CREEDSGeneProcessor**(*records*)
A processor for single gene perturbation experiments in CREEDS.

4.2.4 Custom Knowledge Bases

NDEx CX API (indra.sources.ndex_cx.api)

indra.sources.ndex_cx.api.**process_cx**(*cx_json*, *summary=None*, *require_grounding=True*)
Process a CX JSON object into Statements.

Parameters

- **cx_json** (*list*) – CX JSON object.
- **summary** (*Optional[dict]*) – The network summary object which can be obtained via get_network_summary through the web service. THis contains metadata such as the owner and the creation time of the network.
- **require_grounding** (*bool*) – Whether network nodes lacking grounding information should be included among the extracted Statements (default is True).

Returns Processor containing Statements.

Return type *NdexCxProcessor*

indra.sources.ndex_cx.api.**process_cx_file**(*file_name*, *require_grounding=True*)
Process a CX JSON file into Statements.

Parameters

- **file_name** (*str*) – Path to file containing CX JSON.
- **require_grounding** (*bool*) – Whether network nodes lacking grounding information should be included among the extracted Statements (default is True).

Returns Processor containing Statements.

Return type *NdexCxProcessor*

indra.sources.ndex_cx.api.**process_ndex_network**(*network_id*, *username=None*, *password=None*, *require_grounding=True*)
Process an NDEx network into Statements.

Parameters

- **network_id** (*str*) – NDEx network ID.
- **username** (*str*) – NDEx username.
- **password** (*str*) – NDEx password.
- **require_grounding** (*bool*) – Whether network nodes lacking grounding information should be included among the extracted Statements (default is True).

Returns Processor containing Statements. Returns None if there if the HTTP status code indicates an unsuccessful request.

Return type *NdexCxProcessor*

NDEx CX Processor (indra.sources.ndex_cx.processor)

class indra.sources.ndex_cx.processor.**NdexCxProcessor**(*cx*, *summary=None*, *require_grounding=True*)

The NdexCxProcessor extracts INDRA Statements from Cytoscape CX JSON.

Parameters

- **cx** (*list of dicts*) – JSON content containing the Cytoscape network in CX format.
- **summary** (*Optional[dict]*) – The network summary object which can be obtained via get_network_summary through the web service. THis contains metadata such as the owner and the creation time of the network.

statements
A list of extracted INDRA Statements. Not all edges in the network may be converted into Statements.

Type list

get_agents()
Get list of grounded nodes in the network as Agents.

Returns Only nodes containing sufficient information to be grounded will be contained in this list.

Return type list of Agents

get_node_names()
Get list of all nodes in the network by name.

get_pmids()
Get list of all PMIDs associated with edges in the network.

get_statements()
Convert network edges into Statements.

Returns Converted INDRA Statements.

Return type list of Statements

INDRA Database REST Client (indra.sources.indra_db_rest)

The INDRA database client allows querying a web service that serves content from a database of INDRA Statements collected and pre-assembled from various sources.

Access to the webservice requires a URL (INDRA_DB_REST_URL) and an API key (INDRA_DB_REST_API_KEY), both of which may be placed in your config file or as environment variables. If you do not have these but would like to access the database REST API, please contact the INDRA developers.

API to the INDRA Database REST Service (indra.sources.indra_db_rest.api)

INDRA has been used to generate and maintain a database of causal relations as INDRA Statements. The contents of the INDRA Database can be accessed programmatically through this API.

The API includes three high-level query functions that cover many common use cases:

get_statements(): Get statements by agent information and Statement type, e.g. "Statements with object MEK and type Inhibition" (This query function has a generic name to maintain backward compatibility.)

get_statements_for_paper(): Get Statements based on the papers they are drawn from, for instance "Statements from the paper with PMID 12345".

get_statements_by_hash(): Distinct INDRA Statements are associated with a unique numeric hash. This endpoint can be used to query the database for provenance

Queries with more complex constraints can be made using the query language API in **:py:module:`indra.sources.indra_db_rest.query`** along with this function:

get_statements_from_query(): This function works alongside the Query "language" to execute arbitrary requests for Statements based on statement metadata indexed in the Database.

There are also two functions relating to the submission and retrieval of curations. It is possible to enter feedback the correctness of text-mined Statements, which we call "curations". `submit_curations()` allows you to submit your curations, and *get_curations()* allows you to retrieve existing curations (an API key is required).

Limits, timeouts and threading

Some queries may return a large number of statements, requiring the client to assemble results from multiple successive requests to the REST API. The behavior of the client can be controlled by several parameters to the query functions.

For example, consider the query for Statements whose subject is TNF:

```
>>>
>> from indra.sources.indra_db_rest.api import get_statements
>> p = get_statements("TNF")
>> stmts = p.statements
```

Because there are many Statements associated with TNF, the client will make multiple paged requests to get all the results. The maximum number of Statements returned can be limited using the *limit* argument:

```
>>>
>> p = get_statements("TNF", limit=1000)
>> stmts = p.statements
```

For longer requests the client can work in a background thread after a timeout is reached. This can be done by specifying a timeout (in seconds) using the *timeout* argument. While the client continues retrieval, the first page of the statement results is available in the *statements_sample* attribute:

```
>>>
>> p = get_statements("TNF", timeout=5)
>> some_stmts = p.statements_sample
>>
>> # ...Do some other work...
>>
```

(continues on next page)

(continued from previous page)

```
>> # Wait for the requests to finish before getting the final result.
>> p.wait_until_done()
>> stmts = p.statements
```

Note that the timeout specifies how long the client should block for the result, but that the result will continue to be retrieved until it is completed on a background thread. If desired one can supply a timeout of 0 and get the processor immediately, leaving the entire query to happen in the background.

You can check if the process is still running using the *is_working* method:

```
>>>
>> p = get_statements("TNF", timeout=0)
>> p.is_working()
True
```

If you don't want the client to make multiple paged requests and instead want to get only the results from the first request, you can set "persist" to False (the request job can still be put in the background with *timeout=0*).

```
>>>
>> p = get_statements("TNF", persist=False)
>> stmts = p.statements
```

For additional details on these and other parameters controlling statement retrieval see the function documentation.

Using the Query Language

There are several metadata and data values indexed in the INDRA Database allowing for complex queries. Using the Query language these attributes can be combined in arbitrary ways using logical operators. For example, you may want to find Statements that MEK is inhibited found in papers related to breast cancer and that also have more than 10 evidence:

```
>>>
>> from indra.sources.indra_db_rest.api import get_statements_from_query
>> from indra.sources.indra_db_rest.query import HasAgent, HasType, \
>>     FromMeshIds, HasEvidenceBound
>>
>> query = (HasAgent("MEK", namespace="FPLX") & HasType(["Inhibition"])
>>          & FromMeshIds(["D001943"]) & HasEvidenceBound(["> 10"]))
>>
>> p = get_statements_from_query(query)
>> stmts = p.statements
```

In addition to joining constraints with "&" (an intersection, an "and") as shown above, you can also form unions (a.k.a. "or"s) using "|":

```
>>>
>> query = (
>>     (
>>         HasAgent("MEK", namespace="FPLX")
>>         | HasAgent("MAP2K1", namespace="HGNC-SYMBOL")
>>     )
>>     & HasType(['Inhibition'])
```

(continues on next page)

(continued from previous page)

```
>> )
>>
>> p = get_statements_from_query(query, limit=10)
```

For more details and examples of the Query architecture, see *query*.

Evidence Filtering

Queries can constrain results based on a property of the original evidence text, so anything from the text references (like pmid) to the readers included and whether the evidence is from a reading or a database, can all have an effect on the evidences included in the result. By default, such queries filter not only the statements but also their associated evidence, so that, for example, if you query for Statements from a given paper, the evidences returned with the Statements you queried are only from that paper.

```
>>>
>> p = get_statements_for_papers([('pmid', '20471474'),
>>                                ('pmcid', 'PMC3640704')])
>> all(ev.text_refs['PMID'] == '20471474'
>>     or ev.text_refs['PMCID'] == 'PMC3640704'
>>     for s in p.statements for ev in s.evidence)
True
```

You can deactivate this feature by setting *filter_ev* to False:

```
>>>
>> p = get_statements_for_papers([('pmid', '20471474'),
>>                                ('pmcid', 'PMC3640704')], filter_ev=False)
>> all(ev.text_refs['PMID'] == '20471474'
>>     or ev.text_refs['PMCID'] == 'PMC3640704'
>>     for s in p.statements for ev in s.evidence)
False
```

Curation Submission

Suppose you run a query and get some Statements with some evidence; you look through the results and find an evidence that does not really support the Statement. Using the API it is possible to provide feedback by submitting a curation.

```
>>>
>> from indra.statements import pretty_print_stmts
>> p = get_statements(agents=["TNF"], ev_limit=3, limit=1)
>> pretty_print_stmts(p.statements)
[LIST INDEX: 0] Activation(TNF(), apoptotic process())
================================================================================
EV INDEX: 0       These published reports in their aggregate support that TNFR2
SOURCE: reach     can lower the threshold of bioavailable TNFalpha needed to
PMID: 19774075    cause apoptosis through TNFR1 thus amplifying extrinsic cell
                  death pathways.
--------------------------------------------------------------------------------
EV INDEX: 1       Our results indicate that IE86 inhibits tumor necrosis factor
SOURCE: reach     (TNF)-alpha induced apoptosis and that the anti-apoptotic
```

(continues on next page)

(continued from previous page)

```
PMID: 19502735    activity of this viral protein correlates with its expression
                  levels.
--------------------------------------------------------------------------------
EV INDEX: 2       This relationship between PUFAs and their anti-inflammatory
SOURCE: reach     metabolites and type 1 DM is supported by the observation that
PMID: 28824543    in a mfat-1 transgenic mouse model whose islets contained
                  increased levels of n-3 PUFAs and significantly lower amounts
                  of n-6 PUFAs compared to the wild type, were resistant to
                  apoptosis induced by TNF-alpha, IL-1beta, and gamma-IFN.
--------------------------------------------------------------------------------
>>
>> submit_curation(p.statements[0].get_hash(), "correct", "usr@bogusemail.com",
>>                 pa_json=p.statements[0].to_json(),
>>                 ev_json=p.statements[0].evidence[1].to_json())
{'ref': {'id': 11919}, 'result': 'success'}
```

indra.sources.indra_db_rest.api.**get_statements**(*subject=None*, *object=None*, *agents=None*, *stmt_type=None*, *use_exact_type=False*, *limit=None*, *persist=True*, *timeout=None*, *strict_stop=False*, *ev_limit=10*, *sort_by='ev_count'*, *tries=3*, *use_obtained_counts=False*, *api_key=None*)

Get Statements from the INDRA DB web API matching given agents and type.

You get a *DBQueryStatementProcessor* object, which allow Statements to be loaded in a background thread, providing a sample of the "best" content available promptly in the `sample_statements` attribute, and populates the statements attribute when the paged load is complete. The "best" is determined by the `sort_by` attribute, which may be either 'belief' or 'ev_count' or None.

Parameters

- **subject/object** (*str*) – Optionally specify the subject and/or object of the statements you wish to get from the database. By default, the namespace is assumed to be HGNC gene names, however you may specify another namespace by including "@<namespace>" at the end of the name string. For example, if you want to specify an agent by chebi, you could use "CHEBI:6801@CHEBI", or if you wanted to use the HGNC id, you could use "6871@HGNC".
- **agents** (*list[str]*) – A list of agents, specified in the same manner as subject and object, but without specifying their grammatical position.
- **stmt_type** (*str*) – Specify the types of interactions you are interested in, as indicated by the sub-classes of INDRA's Statements. This argument is *not* case sensitive. If the statement class given has sub-classes (e.g. RegulateAmount has IncreaseAmount and DecreaseAmount), then both the class itself, and its subclasses, will be queried, by default. If you do not want this behavior, set use_exact_type=True. Note that if max_stmts is set, it is possible only the exact statement type will be returned, as this is the first searched. The processor then cycles through the types, getting a page of results for each type and adding it to the quota, until the max number of statements is reached.
- **use_exact_type** (*bool*) – If stmt_type is given, and you only want to search for that specific statement type, set this to True. Default is False.
- **limit** (*Optional[int]*) – Select the maximum number of statements to return. When set less than 500 the effect is much the same as setting persist to false, and will guarantee a faster response. Default is None.

- **persist** (*bool*) – Default is True. When False, if a query comes back limited (not all results returned), just give up and pass along what was returned. Otherwise, make further queries to get the rest of the data (which may take some time).
- **timeout** (*positive int or None*) – If an int, block until the work is done and statements are retrieved, or until the timeout has expired, in which case the results so far will be returned in the response object, and further results will be added in a separate thread as they become available. Block indefinitely until all statements are retrieved. Default is None.
- **strict_stop** (*bool*) – If True, the query will only be given *timeout* time to complete before being abandoned entirely. Otherwise the timeout will simply wait for the thread to join for *timeout* seconds before returning, allowing other work to continue while the query runs in the background. The default is False.
- **ev_limit** (*Optional[int]*) – Limit the amount of evidence returned per Statement. Default is 10.
- **sort_by** (*Optional[str]*) – Str options are currently 'ev_count' or 'belief'. Results will return in order of the given parameter. If None, results will be turned in an arbitrary order.
- **tries** (*Optional[int]*) – Set the number of times to try the query. The database often caches results, so if a query times out the first time, trying again after a timeout will often succeed fast enough to avoid a timeout. This can also help gracefully handle an unreliable connection, if you're willing to wait. Default is 3.
- **use_obtained_counts** (*Optional[bool]*) – If True, evidence counts and source counts are reported based on the actual evidences returned for each statement in this query (as opposed to all existing evidences, even if not all were returned). Default: False
- **api_key** (*Optional[str]*) – Override or use in place of the API key given in the INDRA config file.

Returns processor – An instance of the DBQueryStatementProcessor, which has an attribute `statements` which will be populated when the query/queries are done.

Return type `DBQueryStatementProcessor`

indra.sources.indra_db_rest.api.**get_statements_for_papers**(*ids*, *limit=None*, *ev_limit=10*, *sort_by='ev_count'*, *persist=True*, *timeout=None*, *strict_stop=False*, *tries=3*, *filter_ev=True*, *api_key=None*)

Get Statements extracted from the papers with the given ref ids.

Parameters

- **ids** (*list[str, str]*) – A list of tuples with ids and their type. For example: `[('pmid', '12345'), ('pmcid', 'PMC12345')]` The type can be any one of 'pmid', 'pmcid', 'doi', 'pii', 'manuscript_id', or 'trid', which is the primary key id of the text references in the database.
- **limit** (*Optional[int]*) – Select the maximum number of statements to return. When set less than 500 the effect is much the same as setting persist to false, and will guarantee a faster response. Default is None.
- **ev_limit** (*Optional[int]*) – Limit the amount of evidence returned per Statement. Default is 10.
- **filter_ev** (*bool*) – Indicate whether evidence should have the same filters applied as the statements themselves, where appropriate (e.g. in the case of a filter by paper).
- **sort_by** (*Optional[str]*) – Options are currently 'ev_count' or 'belief'. Results will return in order of the given parameter. If None, results will be turned in an arbitrary order.

- **persist** (*bool*) – Default is True. When False, if a query comes back limited (not all results returned), just give up and pass along what was returned. Otherwise, make further queries to get the rest of the data (which may take some time).
- **timeout** (*positive int or None*) – If an int, return after *timeout* seconds, even if query is not done. Default is None.
- **strict_stop** (*bool*) – If True, the query will only be given *timeout* time to complete before being abandoned entirely. Otherwise the timeout will simply wait for the thread to join for *timeout* seconds before returning, allowing other work to continue while the query runs in the background. The default is False.
- **tries** (*int > 0*) – Set the number of times to try the query. The database often caches results, so if a query times out the first time, trying again after a timeout will often succeed fast enough to avoid a timeout. This can also help gracefully handle an unreliable connection, if you're willing to wait. Default is 3.
- **api_key** (*Optional[str]*) – Override or use in place of the API key given in the INDRA config file.

Returns **processor** – An instance of the DBQueryStatementProcessor, which has an attribute *statements* which will be populated when the query/queries are done.

Return type DBQueryStatementProcessor

indra.sources.indra_db_rest.api.**get_statements_by_hash**(*hash_list*, *limit=None*, *ev_limit=10*, *sort_by='ev_count'*, *persist=True*, *timeout=None*, *strict_stop=False*, *tries=3*, *api_key=None*)

Get Statements from a list of hashes.

Parameters

- **hash_list** (*list[int or str]*) – A list of statement hashes.
- **limit** (*Optional[int]*) – Select the maximum number of statements to return. When set less than 500 the effect is much the same as setting persist to false, and will guarantee a faster response. Default is None.
- **ev_limit** (*Optional[int]*) – Limit the amount of evidence returned per Statement. Default is 10.
- **sort_by** (*Optional[str]*) – Options are currently 'ev_count' or 'belief'. Results will return in order of the given parameter. If None, results will be turned in an arbitrary order.
- **persist** (*bool*) – Default is True. When False, if a query comes back limited (not all results returned), just give up and pass along what was returned. Otherwise, make further queries to get the rest of the data (which may take some time).
- **timeout** (*positive int or None*) – If an int, return after *timeout* seconds, even if query is not done. Default is None.
- **strict_stop** (*bool*) – If True, the query will only be given *timeout* time to complete before being abandoned entirely. Otherwise the timeout will simply wait for the thread to join for *timeout* seconds before returning, allowing other work to continue while the query runs in the background. The default is False.
- **tries** (*int > 0*) – Set the number of times to try the query. The database often caches results, so if a query times out the first time, trying again after a timeout will often succeed fast enough to avoid a timeout. This can also help gracefully handle an unreliable connection, if you're willing to wait. Default is 3.

- **api_key** (*Optional[str]*) – Override or use in place of the API key given in the INDRA config file.

Returns **processor** – An instance of the DBQueryStatementProcessor, which has an attribute *statements* which will be populated when the query/queries are done.

Return type DBQueryStatementProcessor

indra.sources.indra_db_rest.api.**get_statements_from_query**(*query*, *limit=None*, *ev_limit=10*, *sort_by='ev_count'*, *persist=True*, *timeout=None*, *strict_stop=False*, *tries=3*, *filter_ev=True*, *use_obtained_counts=False*, *api_key=None*)

Get Statements using a Query.

Example

```
>>>
>> from indra.sources.indra_db_rest.query import HasAgent, FromMeshIds
>> query = HasAgent("MEK", "FPLX") & FromMeshIds(["D001943"])
>> p = get_statements_from_query(query, limit=100)
>> stmts = p.statements
```

Parameters

- **query** (Query) – The query to be evaluated in return for statements.
- **limit** (*Optional[int]*) – Select the maximum number of statements to return. When set less than 500 the effect is much the same as setting persist to false, and will guarantee a faster response. Default is None.
- **ev_limit** (*Optional[int]*) – Limit the amount of evidence returned per Statement. Default is 10.
- **filter_ev** (*bool*) – Indicate whether evidence should have the same filters applied as the statements themselves, where appropriate (e.g. in the case of a filter by paper).
- **sort_by** (*Optional[str]*) – Options are currently 'ev_count' or 'belief'. Results will return in order of the given parameter. If None, results will be turned in an arbitrary order.
- **persist** (*bool*) – Default is True. When False, if a query comes back limited (not all results returned), just give up and pass along what was returned. Otherwise, make further queries to get the rest of the data (which may take some time).
- **timeout** (*positive int or None*) – If an int, return after `timeout` seconds, even if query is not done. Default is None.
- **strict_stop** (*bool*) – If True, the query will only be given *timeout* time to complete before being abandoned entirely. Otherwise the timeout will simply wait for the thread to join for *timeout* seconds before returning, allowing other work to continue while the query runs in the background. The default is False.
- **use_obtained_counts** (*Optional[bool]*) – If True, evidence counts and source counts are reported based on the actual evidences returned for each statement in this query (as opposed to all existing evidences, even if not all were returned). Default: False
- **tries** (*Optional[int]*) – Set the number of times to try the query. The database often caches results, so if a query times out the first time, trying again after a timeout will often

succeed fast enough to avoid a timeout. This can also help gracefully handle an unreliable connection, if you're willing to wait. Default is 3.

- **api_key** (*Optional[str]*) – Override or use in place of the API key given in the INDRA config file.

Returns **processor** – An instance of the DBQueryStatementProcessor, which has an attribute *statements* which will be populated when the query/queries are done.

Return type DBQueryStatementProcessor

indra.sources.indra_db_rest.api.**submit_curation**(*hash_val*, *tag*, *curator_email*, *text=None*, *source='indra_rest_client'*, *ev_hash=None*, *pa_json=None*, *ev_json=None*, *api_key=None*, *is_test=False*)

Submit a curation for the given statement at the relevant level.

Parameters

- **hash_val** (*int*) – The hash corresponding to the statement.
- **tag** (*str*) – A very short phrase categorizing the error or type of curation, e.g. "grounding" for a grounding error, or "correct" if you are marking a statement as correct.
- **curator_email** (*str*) – The email of the curator.
- **text** (*str*) – A brief description of the problem.
- **source** (*str*) – The name of the access point through which the curation was performed. The default is 'direct_client', meaning this function was used directly. Any higher-level application should identify itself here.
- **ev_hash** (*int*) – A hash of the sentence and other evidence information. Elsewhere referred to as *source_hash*.
- **pa_json** (*None or dict*) – The JSON of a statement you wish to curate. If not given, it may be inferred (best effort) from the given hash.
- **ev_json** (*None or dict*) – The JSON of an evidence you wish to curate. If not given, it cannot be inferred.
- **api_key** (*Optional[str]*) – Override or use in place of the API key given in the INDRA config file.
- **is_test** (*bool*) – Used in testing. If True, no curation will actually be added to the database.

indra.sources.indra_db_rest.api.**get_curations**(*hash_val=None*, *source_hash=None*, *api_key=None*)

Get the curations for a specific statement and evidence.

If neither hash_val nor source_hash are given, all curations will be retrieved. This will require the user to have extra permissions, as determined by their API key.

Parameters

- **hash_val** (*Optional[int]*) – The hash of a statement whose curations you want to retrieve.
- **source_hash** (*Optional[int]*) – The hash generated for a piece of evidence for which you want curations. The *hash_val* must be provided to use the *source_hash*.
- **api_key** (*Optional[str]*) – Override or use in place of the API key given in the INDRA config file.

Returns **curations** – A list of dictionaries containing the curation data.

Return type list

Advanced Query Construction (indra.sources.indra_db_rest.query)

The Query architecture allows the construction of arbitrary queries for content from the INDRA Database.

Specifically, queries constructed using this language of classes is converted into optimized SQL by the INDRA Database REST API. Different classes represent different types of constraints and are named as much as possible to fit together when spoken aloud in English. For example:

```
>>>
>> HasAgent("MEK") & HasAgent("ERK") & HasType(["Phosphorylation"])
```

will find any Statement that has an agent MEK and an agent ERK and has the type phosphorylation.

Query Classes (the building blocks)

Broadly, query classes can be broken into 3 types: queries on the meaning of a Statement, queries on the provenance of a Statement, and queries that combine groups of queries.

Meaning of a Statement:

- *HasAgent*
- *HasType*
- *HasNumAgents*

Provenance of a Statement:

- *HasReadings*
- *HasDatabases*
- *HasSources*
- *HasOnlySource*
- *FromPapers*
- *FromMeshIds*
- *HasNumEvidence*
- *HasEvidenceBound*

Combine Queriers:

- *And*
- *Or*

There is also the special class, the *EmptyQuery* which is useful when programmatically building a query.

Building Nontrivial Queries (how to put the blocks together)

In practice you should not use *And* or *Or* very often but instead make use of the overloaded & and | operators to put Queries together into more complex structures. In addition you can invert a query, i.e., essentially ask for Statements that do *not* meet certain criteria, e.g. "not has readings". This can be accomplished with the overloaded ~ operator, e.g. `~HasReadings()`.

The query class works by representing and producing a particular JSON structure which is recognized by the INDRA Database REST service, where it is translated into a similar but more sophisticated Query language used by the Read-only Database client. The Query class implements the basic methods used to communicate with the REST Service in this way.

Examples

First a couple of examples of the typical usage of a query object (See the *get_statements_from_query* documentation for more usage details):

Example 1: Get statements that have database evidence and have either MEK or MAP2K1 as a name for any of its agents.

```
>>>
>> from indra.sources.indra_db_rest.api import get_statements_from_query
>> from indra.sources.indra_db_rest.query import *
>> q = HasAgent('MEK') | HasAgent('MAP2K1') & HasDatabases()
>> p = get_statements_from_query(q)
>> p.statements
[Activation(MEK(), ERK()),
 Phosphorylation(MEK(), ERK()),
 Activation(MAP2K1(), ERK()),
 Activation(RAF1(), MEK()),
 Phosphorylation(RAF1(), MEK()),
 Phosphorylation(MAP2K1(), ERK()),
 Activation(BRAF(), MEK()),
 Inhibition(2-(2-amino-3-methoxyphenyl)chromen-4-one(), MEK()),
 Activation(MAP2K1(), MAPK1()),
 Activation(MAP2K1(), MAPK3()),
 Phosphorylation(MAP2K1(), MAPK1()),
 Phosphorylation(BRAF(), MEK()),
 Activation(MEK(), MAPK1()),
 Complex(BRAF(), MAP2K1()),
 Phosphorylation(MAP2K1(), MAPK3()),
 Activation(MEK(), MAPK3()),
 Complex(MAP2K1(), RAF1()),
 Activation(RAF1(), MAP2K1()),
 Inhibition(trametinib(), MEK()),
 Phosphorylation(MEK(), MAPK3()),
 Complex(MAP2K1(), MAPK1()),
 Phosphorylation(MEK(), MAPK1()),
 Inhibition(selumetinib(), MEK()),
 Phosphorylation(PAK1(), MAP2K1(), S, 298)]
```

Example 2: Get statements that have an agent MEK and an agent ERK and more than 10 evidence.

```
>>>
>> q = HasAgent('MEK') & HasAgent('ERK') & HasEvidenceBound(["> 10"])
>> p = get_statements_from_query(q)
>> p.statements
[Activation(MEK(), ERK()),
 Phosphorylation(MEK(), ERK()),
 Complex(ERK(), MEK()),
 Inhibition(MEK(), ERK()),
 Dephosphorylation(MEK(), ERK()),
 Complex(ERK(), MEK(), RAF()),
 Phosphorylation(MEK(), ERK(), T),
 Phosphorylation(MEK(), ERK(), Y),
 Activation(MEK(), ERK(mods: (phosphorylation))),
 IncreaseAmount(MEK(), ERK())]
```

Example 3: An example of using the ~ feature.

```
>>>
>> q = HasAgent('MEK', namespace='FPLX') & ~HasAgent('ERK', namespace='FPLX')
>> p = get_statements_from_query(q)
>> p.statements[:10]
[Phosphorylation(None, MEK()),
 Phosphorylation(RAF(), MEK()),
 Activation(RAF(), MEK()),
 Activation(MEK(), MAPK()),
 Inhibition(U0126(), MEK()),
 Inhibition(MEK(), apoptotic process()),
 Activation(MEK(), cell population proliferation()),
 Activation(RAF1(), MEK()),
 Phosphorylation(MEK(), MAPK()),
 Phosphorylation(RAF1(), MEK())]
```

And now an example showing the different methods of the *Query* object:

Example 4: a tour demonstrating key utilities of a query object.

Consider the last query we wrote. You can examine the simple JSON sent to the server:

```
>>>
>> q.to_simple_json()
{'class': 'And',
 'constraint': {'queries': [{'class': 'HasAgent',
    'constraint': {'agent_id': 'MEK',
     'namespace': 'FPLX',
     'role': None,
     'agent_num': None},
    'inverted': False},
   {'class': 'HasAgent',
    'constraint': {'agent_id': 'ERK',
     'namespace': 'FPLX',
     'role': None,
     'agent_num': None},
    'inverted': True}]},
 'inverted': False}
```

Or you can retrieve the more "true" JSON representation that is generated by the server from your simpler query:

```
>>>
>> q.get_query_json()
{'class': 'Intersection',
 'constraint': {'query_list': [{'class': 'HasAgent',
    'constraint': {'_regularized_id': 'MEK',
     'agent_id': 'MEK',
     'agent_num': None,
     'namespace': 'FPLX',
     'role': None},
    'inverted': False},
   {'class': 'HasAgent',
    'constraint': {'_regularized_id': 'ERK',
     'agent_id': 'ERK',
     'agent_num': None,
     'namespace': 'FPLX',
     'role': None},
    'inverted': True}]},
 'inverted': False}
```

And last of all you can retrieve a human readable English description of the query from the server:

```
>>>
>> query_english = q.get_query_english()
>> print("I am finding statements that", query_english)
I am finding statements that do not have an agent where FPLX=ERK and have an
agent where FPLX=MEK
```

class indra.sources.indra_db_rest.query.**Query**

Bases: object

The parent of all query objects.

get(*result_type*, *limit=None*, *sort_by=None*, *offset=None*, *timeout=None*, *n_tries=2*, *api_key=None*, ***other_params*)

Get results from the API of the given type.

Parameters

- **result_type** (*str*) – The options are 'statements', 'interactions', 'relations', 'agents', and 'hashes', indicating the type of result you want.
- **limit** (*Optional[int]*) – The maximum number of statements you want to try and retrieve. The server will by default limit the results, and any value exceeding that limit will be "overruled".
- **sort_by** (*Optional[str]*) – The value can be 'default', 'ev_count', or 'belief'.
- **offset** (*Optional[int]*) – The offset of the query to begin at.
- **timeout** (*Optional[int]*) – The number of seconds to wait for the request to return before giving up. This timeout is applied to each try separately.
- **n_tries** (*Optional[int]*) – The number of times to retry the request before giving up. Each try will have *timeout* seconds to complete before it gives up.
- **api_key** (*str or None*) – Override or use in place of the API key given in the INDRA config file.

- **filter_ev** (*bool*) – (for `result_type='statements'`) Indicate whether evidence should have the same filters applied as the statements themselves, where appropriate (e.g. in the case of a filter by paper).
- **ev_limit** (*int*) – (for `result_type='statements'`) Limit the number of evidence returned per Statement.
- **with_hashes** (*bool*) – (for `result_type='relations'` or `result_type='agents'`) Choose whether the hashes for each Statement be included along with each grouped heading.
- **complexes_covered** (*list[int]*) – (for `result_type='agents'`) A list (or set) of complexes that have already come up in the agent groups returned. This prevents duplication.

get_query_json()
Generate a compiled JSON rep of the query on the server.

get_query_english(*timeout=None*)
Get the string representation of the query.

copy()
Make a copy of the query.

to_simple_json()
Generate the JSON from the object rep.

Return type dict

class indra.sources.indra_db_rest.query.**And**(*queries*)
Bases: *indra.sources.indra_db_rest.query.Query*

The intersection of two queries.

This are generally generated from the use of &, for example:

```
>>>
>> q_and = HashAgent('MEK') & HasAgent('ERK')
```

class indra.sources.indra_db_rest.query.**Or**(*queries*)
Bases: *indra.sources.indra_db_rest.query.Query*

The union of two queries.

These are generally generated from the use of |, for example:

```
>>>
>> q_or = HasOnlySource('reach') | HasOnlySource('medscan')
```

class indra.sources.indra_db_rest.query.**HasAgent**(*agent_id=None*, *namespace='NAME'*, *role=None*, *agent_num=None*)
Bases: *indra.sources.indra_db_rest.query.Query*

Find Statements with the given agent in the given position.

NOTE: At this time 2 agent queries do NOT necessarily imply that the 2 agents are different. For example:

```
>>>
>> HasAgent("MEK") & HasAgent("MEK")
```

will get any Statements that have agent with name MEK, **not** Statements with two agents called MEK. This may change in the future, however in the meantime you can get around this fairly well by specifying the roles:

```
>>>
>> HasAgent("MEK", role="SUBJECT") & HasAgent("MEK", role="OBJECT")
```

Or for a more complicated case, consider a query for Statements where one agent is MEK and the other has namespace FPLX. Naturally any agent labeled as MEK will also have a namespace FPLX (MEK is a famplex identifier), and in general you will not want to constrain which role is MEK and which is the "other" agent. To accomplish this you need to use `|`:

```
>>>
>> (
>>   HasAgent("MEK", role="SUBJECT")
>>   & HasAgent(namespace="FPLX", role="OBJECT")
>> ) | (
>>   HasAgent("MEK", role="OBJECT")
>>   & HasAgent(namespace="FPLX", role="SUBJECT")
>> )
```

Parameters

- **agent_id** (*Optional[str]*) – The ID string naming the agent, for example 'ERK' (FPLX or NAME) or 'plx' (TEXT), and so on. If None, the query must then be constrained by the `namespace`.
- **namespace** (*Optional[str]*) – By default, this is NAME, indicating the agents canonical, grounded, name will be used. Other options include, but are not limited to: AUTO (in which case GILDA will be used to guess the proper grounding of the entity), FPLX (FamPlex), CHEBI, CHEMBL, HGNC, UP (UniProt), and TEXT (for raw text mentions). If `agent_id` is `None`, namespace must be specified and must **not** be NAME, TEXT, or AUTO.
- **role** (*Optional[str]*) – None by default. Options are "SUBJECT", "OBJECT", or "OTHER".
- **agent_num** (*Optionals[int]*) – None by default. The regularized position of the agent in the Statement's list of agents.

class indra.sources.indra_db_rest.query.**FromMeshIds**(*mesh_ids*)

Bases: *indra.sources.indra_db_rest.query.Query*

Get stmts that came from papers annotated with the given Mesh Ids.

Parameters **mesh_ids** (*list*) – A canonical MeSH ID, of the "C" or "D" variety, e.g. "D000135".

class indra.sources.indra_db_rest.query.**HasHash**(*stmt_hashes*)

Bases: *indra.sources.indra_db_rest.query.Query*

Find Statements whose hash is contained in the given list.

Parameters **stmt_hashes** (*list or set or tuple*) – A collection of integers, where each integer is a shallow matches key hash of a Statement (frequently simply called "mk_hash" or "hash")

class indra.sources.indra_db_rest.query.**HasSources**(*sources*)

Bases: *indra.sources.indra_db_rest.query.Query*

Find Statements with support from the given list of sources.

For example, find Statements that have support from both medscan and reach.

Parameters **sources** (*list or set or tuple*) – A collection of strings, each string the canonical name for a source. The result will include statements that have evidence from ALL sources that you include.

class indra.sources.indra_db_rest.query.**HasOnlySource**(*only_source*)
Bases: *indra.sources.indra_db_rest.query.Query*

Find Statements that come exclusively from one source.

For example, find statements that come only from sparser.

Parameters **only_source** (*str*) – The only source that spawned the statement, e.g. signor, or reach.

class indra.sources.indra_db_rest.query.**HasReadings**
Bases: *indra.sources.indra_db_rest.query.Query*

Find Statements with support from readings.

class indra.sources.indra_db_rest.query.**HasDatabases**
Bases: *indra.sources.indra_db_rest.query.Query*

Find Statements with support from Databases.

class indra.sources.indra_db_rest.query.**HasType**(*stmt_types*, *include_subclasses=False*)
Bases: *indra.sources.indra_db_rest.query.Query*

Get Statements with the given type.

For example, you can find Statements that are Phosphorylations or Activations, or you could find all subclasses of RegulateActivity.

Parameters

- **stmt_types** (*set or list or tuple*) – A collection of Strings, where each string is a class name for a type of Statement. Spelling and capitalization are necessary.
- **include_subclasses** (*bool*) – (optional) default is False. If True, each Statement type given in the list will be expanded to include all of its sub classes.

class indra.sources.indra_db_rest.query.**HasNumAgents**(*agent_nums*)
Bases: *indra.sources.indra_db_rest.query.Query*

Get Statements with the given number of agents.

For example, *HasNumAgents([1,3,4])* will return agents with either 2, 3, or 4 agents (the latter two mostly being complexes).

Parameters **agent_nums** (*tuple*) – A list of integers, each indicating a number of agents.

class indra.sources.indra_db_rest.query.**HasNumEvidence**(*evidence_nums*)
Bases: *indra.sources.indra_db_rest.query.Query*

Get Statements with the given number of evidence.

For example, HasNumEvidence([2,3,4]) will return Statements that have either 2, 3, or 4 evidence.

Parameters **evidence_nums** (Tuple[Union[int, str]]) – A list of numbers greater than 0, each indicating a number of evidence.

class indra.sources.indra_db_rest.query.**HasEvidenceBound**(*evidence_bounds*)
Bases: *indra.sources.indra_db_rest.query.Query*

Get Statements with given bounds on their evidence count.

For example, HasEvidenceBound(["< 10", ">= 5"]) will return Statements with less than 10 and as many or more than 5 evidence.

Parameters **evidence_bounds** (Union[Iterable[str], str]) – An iterable (e.g. list) of strings such as "< 2" or ">= 4". The argument of the inequality must be a natural number (0, 1, 2, ...) and the inequality operation must be one of: <, >, <=, >=, ==, !=.

class indra.sources.indra_db_rest.query.**FromPapers**(*paper_list*)
Bases: *indra.sources.indra_db_rest.query.Query*

Get Statements that came from a given list of papers.

Parameters **paper_list** (*list[(<id_type>, <paper_id>)]*) – A list of tuples, where each tuple indicates and id-type (e.g. 'pmid') and an id value for a particular paper.

class indra.sources.indra_db_rest.query.**EmptyQuery**
Bases: *indra.sources.indra_db_rest.query.Query*

A query that is empty.

INDRA Database REST Processor (indra.sources.indra_db_rest.processor)

Retrieving the results of large queries from the INDRA Database REST API generally involves multiple individual calls. The Processor classes defined here manage the retrieval process for results of two types, Statements and Statement hashes. Instances of these Processors are returned by the query functions in *indra.sources.indra_db_rest.api*.

class indra.sources.indra_db_rest.processor.**IndraDBQueryProcessor**(*query*, *limit=None*, *sort_by='ev_count'*, *timeout=None*, *strict_stop=False*, *persist=True*, *tries=3*, *api_key=None*)
Bases: object

The parent of all db query processors.

Parameters

- **query** (Query) – The query to be evaluated in return for statements.
- **limit** (*int or None*) – Select the maximum number of statements to return. When set less than 500 the effect is much the same as setting persist to false, and will guarantee a faster response. Default is None.
- **sort_by** (*str or None*) – Options are currently 'ev_count' or 'belief'. Results will return in order of the given parameter. If None, results will be turned in an arbitrary order.
- **persist** (*bool*) – Default is True. When False, if a query comes back limited (not all results returned), just give up and pass along what was returned. Otherwise, make further queries to get the rest of the data (which may take some time).
- **timeout** (*positive int or None*) – If an int, return after *timeout* seconds, even if query is not done. Default is None.
- **strict_stop** (*bool*) – If True, the query will only be given timeout to complete before being abandoned entirely. Otherwise the timeout will simply wait for the thread to join for *timeout* seconds before returning, allowing other work to continue while the query runs in the background. The default is False. NOTE: in practice, due to overhead, the precision of the timeout is only around +/-0.1 seconds.
- **tries** (*int > 0*) – Set the number of times to try the query. The database often caches results, so if a query times out the first time, trying again after a timeout will often succeed fast enough to avoid a timeout. This can also help gracefully handle an unreliable connection, if you're willing to wait. Default is 3
- **api_key** (*str or None*) – Override or use in place of the API key given in the INDRA config file.

get_ev_counts()
Get a dictionary of evidence counts.

get_belief_scores()
Get a dictionary of belief scores.

get_source_counts()
Get the source counts as a dict per statement hash.

cancel()
Cancel the job, stopping the thread running in the background.

is_working()
Check if the thread is running.

timed_out()
Check if the processor timed out.

wait_until_done(*timeout=None*)
Wait for the background load to complete.

static print_quiet_logs()
Print the logs that were suppressed during the query.

class indra.sources.indra_db_rest.processor.**DBQueryStatementProcessor**(*query*, *limit=None*, *sort_by='ev_count'*, *ev_limit=10*, *filter_ev=True*, *timeout=None*, *strict_stop=False*, *persist=True*, *use_obtained_counts=False*, *tries=3*, *api_key=None*)

Bases: *indra.sources.indra_db_rest.processor.IndraDBQueryProcessor*

A Processor to get Statements from the server.

For information on thread control and other methods, see the docs for *IndraDBQueryProcessor*.

Parameters

- **query** (Query) – The query to be evaluated in return for statements.
- **limit** (*int or None*) – Select the maximum number of statements to return. When set less than 500 the effect is much the same as setting persist to false, and will guarantee a faster response. Default is None.
- **ev_limit** (*int or None*) – Limit the amount of evidence returned per Statement. Default is 100.
- **filter_ev** (*bool*) – Indicate whether evidence should have the same filters applied as the statements themselves, where appropriate (e.g. in the case of a filter by paper).
- **sort_by** (*str or None*) – Options are currently 'ev_count' or 'belief'. Results will return in order of the given parameter. If None, results will be turned in an arbitrary order.
- **persist** (*bool*) – Default is True. When False, if a query comes back limited (not all results returned), just give up and pass along what was returned. Otherwise, make further queries to get the rest of the data (which may take some time).
- **timeout** (*positive int or None*) – If an int, return after *timeout* seconds, even if query is not done. Default is None.

- **strict_stop** (*bool*) – If True, the query will only be given timeout to complete before being abandoned entirely. Otherwise the timeout will simply wait for the thread to join for *timeout* seconds before returning, allowing other work to continue while the query runs in the background. The default is False.
- **use_obtained_counts** (*Optional[bool]*) – If True, evidence counts and source counts are reported based on the actual evidences returned for each statement in this query (as opposed to all existing evidences, even if not all were returned). Default: False
- **tries** (*int > 0*) – Set the number of times to try the query. The database often caches results, so if a query times out the first time, trying again after a timeout will often succeed fast enough to avoid a timeout. This can also help gracefully handle an unreliable connection, if you're willing to wait. Default is 3.
- **api_key** (*str or None*) – Override or use in place of the API key given in the INDRA config file.

get_ev_count_by_hash(*stmt_hash*)
Get the total evidence count for a statement hash.

get_ev_count(*stmt*)
Get the total evidence count for a statement.

get_belief_score_by_hash(*stmt_hash*)
Get the belief score for a statement hash.

get_belief_score_by_stmt(*stmt*)
Get the belief score for a statement.

get_hash_statements_dict()
Return a dict of Statements keyed by hashes.

get_source_count_by_hash(*stmt_hash*)
Get the source counts for a given statement.

get_source_count(*stmt*)
Get the source counts for a given statement.

merge_results(*other_processor*)
Merge the results of this processor with those of another.

class indra.sources.indra_db_rest.processor.**DBQueryHashProcessor**(**args*, ***kwargs*)
Bases: *indra.sources.indra_db_rest.processor.IndraDBQueryProcessor*

A processor to get hashes from the server.

Parameters

- **query** (Query) – The query to be evaluated in return for statements.
- **limit** (*int or None*) – Select the maximum number of statements to return. When set less than 500 the effect is much the same as setting persist to false, and will guarantee a faster response. Default is None.
- **sort_by** (*str or None*) – Options are currently 'ev_count' or 'belief'. Results will return in order of the given parameter. If None, results will be turned in an arbitrary order.
- **persist** (*bool*) – Default is True. When False, if a query comes back limited (not all results returned), just give up and pass along what was returned. Otherwise, make further queries to get the rest of the data (which may take some time).
- **timeout** (*positive int or None*) – If an int, return after *timeout* seconds, even if query is not done. Default is None.

- **tries** (*int > 0*) – Set the number of times to try the query. The database often caches results, so if a query times out the first time, trying again after a timeout will often succeed fast enough to avoid a timeout. This can also help gracefully handle an unreliable connection, if you're willing to wait. Default is 3.

Hypothes.is (indra.sources.hypothesis)

This module implements an API and processor for annotations coming from hypothes.is. Annotations for a given group are obtained and processed either into INDRA Statements or into entity grounding annotations.

Two configurable values (either in the INDRA config file or as an environmental variable) are used. HYPOTHESIS_API_KEY is an API key used to access the hypothes.is API. HYPOTHESIS_GROUP is an optional configuration used to select a specific group of annotations on hypothes.is by default.

Curation tutorial

Go to https://web.hypothes.is/ and create an account, and then create a group in which annotations will be collected. Under Settings, click on Developer to find the API key. Set his API key in the INDRA config file under HYPOTHESIS_API_KEY. Optionally, set the group's ID as HYPOTHESIS_GROUP in the INDRA config file. (Note that both these values can also be set as environmental variables.) Next, install the hypothes.is browser plug-in and log in.

Curating Statements

To curate text from a website with the intention of creating one or more INDRA Statements, select some text and create a new annotation using the hypothes.is browser plug-in. The content of the annotation consists of one or more lines. The first line should contain one or more English sentences describing the mechanism(s) that will be represented as an INDRA Statement (e.g., AMPK activates STAT3) based on the selected text. Each subsequent line of the annotation is assumed to be a context annotation. These lines are of the form "<context type>: <context text>" where <context type> can be one of: Cell type, Cell line, Disease, Organ, Location, Species, and <context text> is the text describing the context, e.g., lysosome, liver, prostate cancer, etc.

The annotation should also be tagged with *indra* (though by default, if no tags are given, the processor assumes that the given annotation is an INDRA Statement annotation).

Curating grounding

Generally, grounding annotations are only needed if INDRA's current resources (reading systems, grounding mapping, Gilda, etc.) don't contain a given synonym for an entity of interest.

With the hypothes.is browser plug-in, select some text on a website that contains lexical information about an entity or concept of interest. The conctent of the new annotation can contain one or more lines with identical syntax as follows: [text to ground] -> <db_name1>:<db_id1>|<db_name2>:<db_id2>|... In each case, db_name is a grounding database name space such as HGNC or CHEBI, and db_id is a value within that namespace such as 1097 or CHEBI:63637. Example: [AMPK] -> FPLX:AMPK.

The annotation needs to be tagged with *gilda* for the processor to know that it needs to be interpreted as a grounding annotation.

Hypothes.is API (indra.sources.hypothesis.api)

indra.sources.hypothesis.api.**get_annotations**(*group=None*)

Return annotations in hypothes.is in a given group.

Parameters **group** (*Optional[str]*) – The hypothesi.is key of the group (not its name). If not given, the HYPOTHESIS_GROUP configuration in the config file or an environmental variable is used.

indra.sources.hypothesis.api.**process_annotations**(*group=None*, *reader=None*, *grounder=None*)

Process annotations in hypothes.is in a given group.

Parameters

- **group** (*Optional[str]*) – The hypothesi.is key of the group (not its name). If not given, the HYPOTHESIS_GROUP configuration in the config file or an environmental variable is used.
- **reader** (*Optional[None, str, Callable[[str], Processor]]*) – A handle for a function which takes a single str argument (text to process) and returns a processor object with a statements attribute containing INDRA Statements. By default, the REACH reader's process_text function is used with default parameters. Note that if the function requires extra parameters other than the input text, functools.partial can be used to set those. Can be alternatively set to `indra.sources.bel.process_text()` by using the string "bel".
- **grounder** (*Optional[function]*) – A handle for a function which takes a positional str argument (entity text to ground) and an optional context key word argument and returns a list of objects matching the structure of gilda.grounder.ScoredMatch. By default, Gilda's ground function is used for grounding.

Returns A HypothesisProcessor object which contains a list of extracted INDRA Statements in its statements attribute, and a list of extracted grounding curations in its groundings attribute.

Return type *HypothesisProcessor*

Example

Process all annotations that have been written in BEL with:

```
from indra.sources import hypothesis
processor = hypothesis.process_annotations(group='Z8RNqokY', reader='bel')
processor.statements
# returns: [Phosphorylation(AKT(), PCGF2(), T, 334)]
```

If this example doesn't work, try joining the group with this link: https://hypothes.is/groups/Z8RNqokY/cthoyt-bel.

indra.sources.hypothesis.api.**upload_annotation**(*url*, *annotation*, *target_text=None*, *tags=None*, *group=None*)

Upload an annotation to hypothes.is.

Parameters

- **url** (*str*) – The URL of the resource being annotated.
- **annotation** (*str*) – The text content of the annotation itself.
- **target_text** (*Optional[str]*) – The specific span of text that the annotation applies to.
- **tags** (*list[str]*) – A list of tags to apply to the annotation.

- **group** (*Optional[str]*) – The hypothesi.is key of the group (not its name). If not given, the HYPOTHESIS_GROUP configuration in the config file or an environmental variable is used.

Returns The full response JSON from the web service.

Return type json

indra.sources.hypothesis.api.**upload_statement_annotation**(*stmt*, *annotate_agents=True*)
Construct and upload all annotations for a given INDRA Statement.

Parameters

- **stmt** (*indra.statements.Statement*) – An INDRA Statement.
- **annotate_agents** (*Optional[bool]*) – If True, the agents in the annotation text are linked to outside databases based on their grounding. Default: True

Returns A list of annotation structures that were uploaded to hypothes.is.

Return type list of dict

Hypothes.is Processor (indra.sources.hypothesis.processor)

class indra.sources.hypothesis.processor.**HypothesisProcessor**(*annotations*, *reader=None*, *grounder=None*)
Processes hypothes.is annotations into INDRA Statements or groundings.

Parameters

- **annotations** (*list[dict]*) – A list of annotations fetched from hypothes.is in JSON-deserialized form represented as a list of dicts.
- **reader** (*Union[None, str, Callable[[str],Processor]]*) – A handle for a function which takes a single str argument (text to process) and returns a processor object with a statements attribute containing INDRA Statements. By default, the REACH reader's process_text function is used with default parameters. Note that if the function requires extra parameters other than the input text, functools.partial can be used to set those.
- **grounder** (*Optional[function]*) – A handle for a function which takes a positional str argument (entity text to ground) and an optional context key word argument and returns a list of objects matching the structure of gilda.grounder.ScoredMatch. By default, Gilda's ground function is used for grounding.

statements
A list of INDRA Statements extracted from the given annotations.

Type list[indra.statements.Statement]

groundings
A dict of entity text keys with an associated dict of grounding references.

Type dict

extract_groundings()
Sets groundings attribute to list of extracted groundings.

extract_statements()
Sets statements attribute to list of extracted INDRA Statements.

static groundings_from_annotation(*annotation*)
Return a dict of groundings from a single annotation.

stmts_from_annotation(*annotation*)
Return a list of Statements extracted from a single annotation.

indra.sources.hypothesis.processor.**get_text_refs**(*url*)
Return the parsed out text reference dict from an URL.

indra.sources.hypothesis.processor.**parse_context_entry**(*entry*, *grounder*, *sentence=None*)
Return a dict of context type and object processed from an entry.

indra.sources.hypothesis.processor.**parse_grounding_entry**(*entry*)
Return a dict representing single grounding curation entry string.

Biofactoid (indra.sources.biofactoid)

This module implements an interface to Biofactoid (https://biofactoid.org/) which contains interactions curated from publications by authors. Documents are retrieved from the web and processed into INDRA Statements.

Biofactoid API (indra.sources.biofactoid.api)

indra.sources.biofactoid.api.**process_from_web**(*url=None*)
Process BioFactoid documents from the web.

Parameters **url** (*Optional[str]*) – The URL for the web service endpoint which contains all the document data.

Returns A processor which contains extracted INDRA Statements in its statements attribute.

Return type *BioFactoidProcessor*

indra.sources.biofactoid.api.**process_json**(*biofactoid_json*)
Process BioFactoid JSON.

Parameters **biofactoid_json** (*json*) – The BioFactoid JSON object to process.

Returns A processor which contains extracted INDRA Statements in its statements attribute.

Return type *BioFactoidProcessor*

Biofactoid Processor (indra.sources.biofactoid.processor)

class indra.sources.biofactoid.processor.**BioFactoidProcessor**(*biofactoid_json*)
Processor which extracts INDRA Statements from BioFactoid JSON.

Parameters **biofactoid_json** (*json*) – BioFactoid JSON to process.

statements
A list of INDRA Statements extracted from the BioFactoid JSON.

Type list[indra.statements.Statement]

MINERVA (indra.sources.minerva)

This module implements extracting INDRA Statements from COVID-19 Disease Map models (https://covid19map.elixir-luxembourg.org/minerva/). Currently it supports a processor that extracts statements from SIF export of the models.

More information about COVID-19 Disease Map project can be found at https://covid.pages.uni.lu

MINERVA Source API (indra.sources.minerva.api)

indra.sources.minerva.api.**process_file**(*filename*, *model_id*, *map_name='covid19map'*)
Get statements by processing a single local SIF file.

Parameters

- **filename** (*str*) – A name (or path) of a local SIF file to process.
- **model_id** (*int*) – ID of a model corresponding to file content. Model ID is needed to find relevant references.
- **map_name** (*str*) – A name of a disease map to process.

Returns **sp** – An instance of a SifProcessor with extracted INDRA statements.

Return type indra.source.minerva.SifProcessor

indra.sources.minerva.api.**process_files**(*ids_to_filenames*, *map_name='covid19map'*)
Get statements by processing one or more local SIF files.

Parameters

- **ids_to_file_names** (*dict*) – A dictionary mapping model IDs to files containing model content as SIF. Model IDs are needed to find relevant references.
- **map_name** (*str*) – A name of a disease map to process.

Returns **sp** – An instance of a SifProcessor with extracted INDRA statements.

Return type indra.source.minerva.SifProcessor

indra.sources.minerva.api.**process_from_web**(*filenames='all'*, *map_name='covid19map'*)
Get statements by processing remote SIF files.

Parameters

- **filenames** (*list or str('all')*) – Filenames for models that need to be processed (for full list of available models see https://git-r3lab.uni.lu/covid/models/-/tree/master/ Executable%20Modules/SBML_qual_build/sif). If set to 'all' (default), then all available models will be processed.
- **map_name** (*str*) – A name of a disease map to process.

Returns **sp** – An instance of a SifProcessor with extracted INDRA statements.

Return type indra.source.minerva.SifProcessor

MINERVA SIF Processor (indra.sources.minerva.processor)

class indra.sources.minerva.processor.**SifProcessor**(*model_id_to_sif_strs*, *map_name='covid19map'*)
Processor that extracts INDRA Statements from SIF strings.

Parameters

- **model_id_to_sif_strs** (*dict*) – A dictionary mapping a model ID (int) to a list of strings in SIF format. Example: {799: ['csa2 POSITIVE sa9', 'csa11 NEGATIVE sa30']}
- **map_name** (*str*) – A name of a disease map to process.

statements
A list of INDRA Statements extracted from the SIF strings.

Type list[indra.statements.Statement]

indra.sources.minerva.processor.**get_agent**(*element_id*, *ids_to_refs*, *complex_members*)
Get an agent for a MINERVA element.

Parameters

- **element_id** (*str*) – ID of an element used in MINERVA API and raw SIF files.
- **ids_to_refs** (*dict*) – A dictionary mapping element IDs to MINERVA provided references. Note that this mapping is unique per model (same IDs can be mapped to different refs in different models).
- **complex_members** (*dict*) – A dictionary mapping element ID of a complex element to element IDs of its members.

Returns **agent** – INDRA agent created from given refs.

Return type *indra.statements.agent.Agent*

indra.sources.minerva.processor.**get_agent_from_refs**(*db_refs*)
Get an agent given its db_refs.

indra.sources.minerva.processor.**get_family**(*agents*)
Get a FamPlex family if all of its members are given.

4.2.5 Utilities

Processor for remote INDRA JSON files.

class indra.sources.utils.**Processor**
A base class for processors.

classmethod cli()
Run the CLI for this processor.

Return type None

extract_statements()
Extract statements from the remote JSON file.

Return type List[*Statement*]

classmethod get_cli()
Get the CLI for this processor.

Return type Command

class indra.sources.utils.**RemoteProcessor**(*url*)
A processor for INDRA JSON file to be retrieved by URL.

Parameters **url** (*str*) – The URL of the INDRA JSON file to load

extract_statements()
Extract statements from the remote JSON file.

Return type List[*Statement*]

print_summary()
Print a summary of the statements.

Return type None

property statements: List[*indra.statements.statements.Statement*]
The extracted statements.

Return type List[*Statement*]

url: str
The URL of the data

4.3 Database clients (indra.databases)

This module implements a number of clients for accessing and using resources from biomedical entity databases and other third-party web services that INDRA uses. Many of the resources these clients use are loaded from resource files in the indra.resources module, in many cases also providing access to web service endpoints.

4.3.1 identifiers.org mappings and URLs (indra.databases.identifiers)

indra.databases.identifiers.**ensure_chebi_prefix**(*chebi_id*)
Return a valid CHEBI ID that has the appropriate CHEBI: prefix.

indra.databases.identifiers.**ensure_chembl_prefix**(*chembl_id*)
Return a valid CHEMBL ID that has the appropriate CHEMBL prefix.

indra.databases.identifiers.**ensure_prefix**(*db_ns*, *db_id*, *with_colon=True*)
Return a valid ID that has the given namespace embedded.

This is useful for namespaces such as CHEBI, GO or BTO that require the namespace to be part of the ID. Note that this function always ensures that the given db_ns is embedded in the ID and can handle the case whene it's already present.

Parameters

- **db_ns** (*str*) – A namespace.
- **db_id** (*str*) – An ID within that namespace which should have the namespace as a prefix in it.
- **with_colon** (*Optional[bool]*) – If True, the namespace prefix is followed by a colon in the ID (e.g., CHEBI:12345). Otherwise, no colon is added (e.g., CHEMBL1234). Default: True

indra.databases.identifiers.**ensure_prefix_if_needed**(*db_ns*, *db_id*)
Return an ID ensuring a namespace prefix if known to be needed.

Parameters

- **db_ns** (str) – The namespace associated with the identifier.
- **db_id** (str) – The original identifier.

Return type str

Returns The identifier with namespace embedded if needed.

indra.databases.identifiers.**get_identifiers_ns**(*db_name*)
Map an INDRA namespace to an identifiers.org namespace when possible.

Example: this can be used to map 'UP' to 'uniprot'.

Parameters **db_name** (*str*) – An INDRA namespace to map to identifiers.org

Returns An identifiers.org namespace or None if not available.

Return type str or None

indra.databases.identifiers.**get_identifiers_url**(*db_name*, *db_id*)
Return an identifiers.org URL for a given database name and ID.

Parameters

- **db_name** (*str*) – An internal database name: HGNC, UP, CHEBI, etc.
- **db_id** (*str*) – An identifier in the given database.

Returns **url** – An identifiers.org URL corresponding to the given database name and ID.

Return type str

indra.databases.identifiers.**get_ns_from_identifiers**(*identifiers_ns*)
"Return a namespace compatible with INDRA from an identifiers namespace.

For example, this function can be used to map 'uniprot' to 'UP'.

Parameters **identifiers_ns** (*str*) – An identifiers.org standard namespace.

Returns The namespace compatible with INDRA's internal representation or None if the given namespace isn't an identifiers.org standard.

Return type str or None

indra.databases.identifiers.**get_ns_id_from_identifiers**(*identifiers_ns*, *identifiers_id*)
Return a namespace/ID pair compatible with INDRA from identifiers.

Parameters

- **identifiers_ns** (*str*) – An identifiers.org standard namespace.
- **identifiers_id** (*str*) – An identifiers.org standard ID in the given namespace.

Returns A namespace and ID that are valid in INDRA db_refs.

Return type (str, str)

indra.databases.identifiers.**get_url_prefix**(*db_name*)
Return the URL prefix for a given namespace.

indra.databases.identifiers.**namespace_embedded**(*db_ns*)
Return true if this namespace requires IDs to have namespace embedded.

This function first maps the given namespace to an identifiers.org namespace and then checks the registry to see if namespaces need to be embedded in IDs. If yes, it returns True. If not, or the ID can't be mapped to identifiers.org, it returns False

Parameters **db_ns** (str) – The namespace to check.

Return type bool

Returns True if the namespace is known to be embedded in IDs of this namespace. False if unknown or known not to be embedded.

indra.databases.identifiers.**parse_identifiers_url**(*url*)
Retrieve database name and ID given the URL.

Parameters **url** (*str*) – An identifiers.org URL to parse.

Returns

- **db_name** (*str*) – An internal database name: HGNC, UP, CHEBI, etc. corresponding to the given URL.
- **db_id** (*str*) – An identifier in the database.

4.3.2 HGNC client (indra.databases.hgnc_client)

indra.databases.hgnc_client.**get_current_hgnc_id**(*hgnc_name*)
Return HGNC ID(s) corresponding to a current or outdated HGNC symbol.

Parameters **hgnc_name** (*str*) – The HGNC symbol to be converted, possibly an outdated symbol.

Returns If there is a single HGNC ID corresponding to the given current or outdated HGNC symbol, that ID is returned as a string. If the symbol is outdated and maps to multiple current IDs, a list of these IDs is returned. If the given name doesn't correspond to either a current or an outdated HGNC symbol, None is returned.

Return type str or list of str or None

indra.databases.hgnc_client.**get_ensembl_id**(*hgnc_id*)
Return the Ensembl ID corresponding to the given HGNC ID.

Parameters **hgnc_id** (*str*) – The HGNC ID to be converted. Note that the HGNC ID is a number that is passed as a string. It is not the same as the HGNC gene symbol.

Returns **ensembl_id** – The Ensembl ID corresponding to the given HGNC ID.

Return type str

indra.databases.hgnc_client.**get_entrez_id**(*hgnc_id*)
Return the Entrez ID corresponding to the given HGNC ID.

Parameters **hgnc_id** (*str*) – The HGNC ID to be converted. Note that the HGNC ID is a number that is passed as a string. It is not the same as the HGNC gene symbol.

Returns **entrez_id** – The Entrez ID corresponding to the given HGNC ID.

Return type str

indra.databases.hgnc_client.**get_gene_type**(*hgnc_id*)
Return the locus type of the genve with the given HGNC ID.

See more under Locus type at https://www.genenames.org/help/symbol-report/#!/#tocAnchor-1-2

Parameters **hgnc_id** (str) – The HGNC ID of the gene to get the locus type of.

Return type Optional[str]

Returns The locus type of the given gene.

indra.databases.hgnc_client.**get_hgnc_entry**(*hgnc_id*)
Return the HGNC entry for the given HGNC ID from the web service.

Parameters **hgnc_id** (*str*) – The HGNC ID to be converted.

Returns **xml_tree** – The XML ElementTree corresponding to the entry for the given HGNC ID.

Return type ElementTree

indra.databases.hgnc_client.**get_hgnc_from_ensembl**(*ensembl_id*)
Return the HGNC ID corresponding to the given Ensembl ID.

Parameters **ensembl_id** (*str*) – The Ensembl ID to be converted, a number passed as a string.

Returns **hgnc_id** – The HGNC ID corresponding to the given Ensembl ID.

Return type str

indra.databases.hgnc_client.**get_hgnc_from_entrez**(*entrez_id*)
Return the HGNC ID corresponding to the given Entrez ID.

Parameters **entrez_id** (*str*) – The Entrez ID to be converted, a number passed as a string.

Returns **hgnc_id** – The HGNC ID corresponding to the given Entrez ID.

Return type str

indra.databases.hgnc_client.**get_hgnc_from_mouse**(*mgi_id*)
Return the HGNC ID corresponding to the given MGI mouse gene ID.

Parameters **mgi_id** (*str*) – The MGI ID to be converted. Example: “2444934”

Returns **hgnc_id** – The HGNC ID corresponding to the given MGI ID.

Return type str

indra.databases.hgnc_client.**get_hgnc_from_rat**(*rgd_id*)
Return the HGNC ID corresponding to the given RGD rat gene ID.

Parameters **rgd_id** (*str*) – The RGD ID to be converted. Example: “1564928”

Returns **hgnc_id** – The HGNC ID corresponding to the given RGD ID.

Return type str

indra.databases.hgnc_client.**get_hgnc_id**(*hgnc_name*)
Return the HGNC ID corresponding to the given HGNC symbol.

Parameters **hgnc_name** (*str*) – The HGNC symbol to be converted. Example: BRAF

Returns **hgnc_id** – The HGNC ID corresponding to the given HGNC symbol.

Return type str

indra.databases.hgnc_client.**get_hgnc_name**(*hgnc_id*)
Return the HGNC symbol corresponding to the given HGNC ID.

Parameters **hgnc_id** (*str*) – The HGNC ID to be converted.

Returns **hgnc_name** – The HGNC symbol corresponding to the given HGNC ID.

Return type str

indra.databases.hgnc_client.**get_mouse_id**(*hgnc_id*)
Return the MGI mouse ID corresponding to the given HGNC ID.

Parameters **hgnc_id** (*str*) – The HGNC ID to be converted. Example: “”

Returns **mgi_id** – The MGI ID corresponding to the given HGNC ID.

Return type str

indra.databases.hgnc_client.**get_rat_id**(*hgnc_id*)
Return the RGD rat ID corresponding to the given HGNC ID.

Parameters **hgnc_id** (*str*) – The HGNC ID to be converted. Example: “”

Returns **rgd_id** – The RGD ID corresponding to the given HGNC ID.

Return type str

indra.databases.hgnc_client.**get_uniprot_id**(*hgnc_id*)
Return the UniProt ID corresponding to the given HGNC ID.

Parameters **hgnc_id** (*str*) – The HGNC ID to be converted. Note that the HGNC ID is a number that is passed as a string. It is not the same as the HGNC gene symbol.

Returns **uniprot_id** – The UniProt ID corresponding to the given HGNC ID.

Return type str

indra.databases.hgnc_client.**is_kinase**(*gene_name*)
Return True if the given gene name is a kinase.

Parameters **gene_name** (*str*) – The HGNC gene symbol corresponding to the protein.

Returns True if the given gene name corresponds to a kinase, False otherwise.

Return type bool

indra.databases.hgnc_client.**is_phosphatase**(*gene_name*)
Return True if the given gene name is a phosphatase.

Parameters **gene_name** (*str*) – The HGNC gene symbol corresponding to the protein.

Returns True if the given gene name corresponds to a phosphatase, False otherwise.

Return type bool

indra.databases.hgnc_client.**is_transcription_factor**(*gene_name*)
Return True if the given gene name is a transcription factor.

Parameters **gene_name** (*str*) – The HGNC gene symbol corresponding to the protein.

Returns True if the given gene name corresponds to a transcription factor, False otherwise.

Return type bool

4.3.3 Uniprot client (indra.databases.uniprot_client)

4.3.4 ChEBI client (indra.databases.chebi_client)

indra.databases.chebi_client.**get_chebi_entry_from_web**(*chebi_id*)
Return a ChEBI entry corresponding to a given ChEBI ID using a REST API.

Parameters **chebi_id** (*str*) – The ChEBI ID whose entry is to be returned.

Returns An ElementTree element representing the ChEBI entry.

Return type xml.etree.ElementTree.Element

indra.databases.chebi_client.**get_chebi_id_from_cas**(*cas_id*)
Return a ChEBI ID corresponding to the given CAS ID.

Parameters **cas_id** (*str*) – The CAS ID to be converted.

Returns **chebi_id** – The ChEBI ID corresponding to the given CAS ID. If the lookup fails, None is returned.

Return type str

indra.databases.chebi_client.**get_chebi_id_from_chembl**(*chembl_id*)
Return a ChEBI ID from a given ChEBML ID.

Parameters **chembl_id** (*str*) – ChEBML ID to be converted.

Returns **chebi_id** – ChEBI ID corresponding to the given ChEBML ID. If the lookup fails, None is returned.

Return type str

indra.databases.chebi_client.**get_chebi_id_from_hmdb**(*hmdb_id*)
Return the ChEBI ID corresponding to an HMDB ID.

Parameters **hmdb_id** (*str*) – An HMDB ID.

Returns The ChEBI ID that the given HMDB ID maps to or None if no mapping was found.

Return type str

indra.databases.chebi_client.**get_chebi_id_from_name**(*chebi_name*)
Return a ChEBI ID corresponding to the given ChEBI name.

Parameters **chebi_name** (*str*) – The ChEBI name whose ID is to be returned.

Returns **chebi_id** – The ID corresponding to the given ChEBI name. If the lookup fails, None is returned.

Return type str

indra.databases.chebi_client.**get_chebi_id_from_pubchem**(*pubchem_id*)
Return the ChEBI ID corresponding to a given Pubchem ID.

Parameters **pubchem_id** (*str*) – Pubchem ID to be converted.

Returns **chebi_id** – ChEBI ID corresponding to the given Pubchem ID. If the lookup fails, None is returned.

Return type str

indra.databases.chebi_client.**get_chebi_name_from_id**(*chebi_id*, *offline=True*)
Return a ChEBI name corresponding to the given ChEBI ID.

Parameters

- **chebi_id** (*str*) – The ChEBI ID whose name is to be returned.
- **offline** (*Optional[bool]*) – If False, the ChEBI web service is invoked in case a name mapping could not be found in the local resource file. Default: True

Returns **chebi_name** – The name corresponding to the given ChEBI ID. If the lookup fails, None is returned.

Return type str

indra.databases.chebi_client.**get_chebi_name_from_id_web**(*chebi_id*)
Return a ChEBI name corresponding to a given ChEBI ID using a REST API.

Parameters **chebi_id** (*str*) – The ChEBI ID whose name is to be returned.

Returns **chebi_name** – The name corresponding to the given ChEBI ID. If the lookup fails, None is returned.

Return type str

indra.databases.chebi_client.**get_chembl_id**(*chebi_id*)
Return a ChEMBL ID from a given ChEBI ID.

Parameters **chebi_id** (*str*) – ChEBI ID to be converted.

Returns **chembl_id** – ChEMBL ID corresponding to the given ChEBI ID. If the lookup fails, None is returned.

Return type str

indra.databases.chebi_client.**get_inchi_key**(*chebi_id*)
Return an InChIKey corresponding to a given ChEBI ID using a REST API.

Parameters **chebi_id** (*str*) – The ChEBI ID whose InChIKey is to be returned.

Returns The InChIKey corresponding to the given ChEBI ID. If the lookup fails, None is returned.

Return type str

indra.databases.chebi_client.**get_primary_id**(*chebi_id*)
Return the primary ID corresponding to a ChEBI ID.

Note that if the provided ID is a primary ID, it is returned unchanged.

Parameters **chebi_id** (*str*) – The ChEBI ID that should be mapped to its primary equivalent.

Returns The primary ChEBI ID or None if the provided ID is neither primary nor a secondary ID with a primary mapping.

Return type str or None

indra.databases.chebi_client.**get_pubchem_id**(*chebi_id*)
Return the PubChem ID corresponding to a given ChEBI ID.

Parameters **chebi_id** (*str*) – ChEBI ID to be converted.

Returns **pubchem_id** – PubChem ID corresponding to the given ChEBI ID. If the lookup fails, None is returned.

Return type str

indra.databases.chebi_client.**get_specific_id**(*chebi_ids*)
Return the most specific ID in a list based on the hierarchy.

Parameters **chebi_ids** (*list of str*) – A list of ChEBI IDs some of which may be hierarchically related.

Returns The first ChEBI ID which is at the most specific level in the hierarchy with respect to the input list.

Return type str

4.3.5 Cell type context client (indra.databases.context_client)

indra.databases.context_client.**get_mutations**(*gene_names*, *cell_types*)
Return protein amino acid changes in given genes and cell types.

Parameters

- **gene_names** (*list*) – HGNC gene symbols for which mutations are queried.
- **cell_types** (*list*) – List of cell type names in which mutations are queried. The cell type names follow the CCLE database conventions.

  Example: LOXIMVI_SKIN, BT20_BREAST

Returns **res** – A dictionary keyed by cell line, which contains another dictionary that is keyed by gene name, with a list of amino acid substitutions as values.

Return type dict[dict[list]]

indra.databases.context_client.**get_protein_expression**(*gene_names*, *cell_types*)
Return the protein expression levels of genes in cell types.

Parameters

- **gene_names** (*list*) – HGNC gene symbols for which expression levels are queried.
- **cell_types** (*list*) – List of cell type names in which expression levels are queried. The cell type names follow the CCLE database conventions.

  Example: LOXIMVI_SKIN, BT20_BREAST

Returns **res** – A dictionary keyed by cell line, which contains another dictionary that is keyed by gene name, with estimated protein amounts as values.

Return type dict[dict[float]]

4.3.6 NDEx client (indra.databases.ndex_client)

indra.databases.ndex_client.**create_network**(*cx_str*, *ndex_cred=None*, *private=True*)
Creates a new NDEx network of the assembled CX model.

To upload the assembled CX model to NDEx, you need to have a registered account on NDEx (http://ndexbio.org/) and have the *ndex* python package installed. The uploaded network is private by default.

Parameters **ndex_cred** (*dict*) – A dictionary with the following entries: 'user': NDEx user name 'password': NDEx password

Returns **network_id** – The UUID of the NDEx network that was created by uploading the assembled CX model.

Return type str

indra.databases.ndex_client.**get_default_ndex_cred**(*ndex_cred*)
Gets the NDEx credentials from the dict, or tries the environment if None

indra.databases.ndex_client.**send_request**(*ndex_service_url*, *params*, *is_json=True*, *use_get=False*)
Send a request to the NDEx server.

Parameters

- **ndex_service_url** (*str*) – The URL of the service to use for the request.
- **params** (*dict*) – A dictionary of parameters to send with the request. Parameter keys differ based on the type of request.

- **is_json** (*bool*) – True if the response is in json format, otherwise it is assumed to be text. Default: False
- **use_get** (*bool*) – True if the request needs to use GET instead of POST.

Returns **res** – Depending on the type of service and the is_json parameter, this function either returns a text string or a json dict.

Return type str

indra.databases.ndex_client.**set_style**(*network_id*, *ndex_cred=None*, *template_id=None*)
Set the style of the network to a given template network's style

Parameters

- **network_id** (*str*) – The UUID of the NDEx network whose style is to be changed.
- **ndex_cred** (*dict*) – A dictionary of NDEx credentials.
- **template_id** (*Optional[str]*) – The UUID of the NDEx network whose style is used on the network specified in the first argument.

indra.databases.ndex_client.**update_network**(*cx_str*, *network_id*, *ndex_cred=None*)
Update an existing CX network on NDEx with new CX content.

Parameters

- **cx_str** (*str*) – String containing the CX content.
- **network_id** (*str*) – UUID of the network on NDEx.
- **ndex_cred** (*dict*) – A dictionary with the following entries: 'user': NDEx user name 'password': NDEx password

4.3.7 cBio portal client (indra.databases.cbio_client)

indra.databases.cbio_client.**get_cancer_studies**(*study_filter=None*)
Return a list of cancer study identifiers, optionally filtered.

There are typically multiple studies for a given type of cancer and a filter can be used to constrain the returned list.

Parameters **study_filter** (*Optional[str]*) – A string used to filter the study IDs to return. Example: "paad"

Returns **study_ids** – A list of study IDs. For instance "paad" as a filter would result in a list of study IDs with paad in their name like "paad_icgc", "paad_tcga", etc.

Return type list[str]

indra.databases.cbio_client.**get_cancer_types**(*cancer_filter=None*)
Return a list of cancer types, optionally filtered.

Parameters **cancer_filter** (*Optional[str]*) – A string used to filter cancer types. Its value is the name or part of the name of a type of cancer. Example: "melanoma", "pancreatic", "non-small cell lung"

Returns **type_ids** – A list of cancer types matching the filter. Example: for cancer_filter="pancreatic", the result includes "panet" (neuro-endocrine) and "paad" (adenocarcinoma)

Return type list[str]

indra.databases.cbio_client.**get_case_lists**(*study_id*)
Return a list of the case set ids for a particular study.

TAKE NOTE the "case_list_id" are the same thing as "case_set_id" Within the data, this string is referred to as a "case_list_id". Within API calls it is referred to as a 'case_set_id'. The documentation does not make this explicitly clear.

Parameters **study_id** (*str*) – The ID of the cBio study. Example: 'cellline_ccle_broad' or 'paad_icgc'

Returns **case_set_ids** – A dict keyed to cases containing a dict keyed to genes containing int

Return type dict[dict[int]]

indra.databases.cbio_client.**get_ccle_cna**(*gene_list*, *cell_lines*)
Return a dict of CNAs in given genes and cell lines from CCLE.

CNA values correspond to the following alterations

-2 = homozygous deletion

-1 = hemizygous deletion

0 = neutral / no change

1 = gain

2 = high level amplification

Parameters

- **gene_list** (*list[str]*) – A list of HGNC gene symbols to get mutations in
- **cell_lines** (*list[str]*) – A list of CCLE cell line names to get mutations for.

Returns **profile_data** – A dict keyed to cases containing a dict keyed to genes containing int

Return type dict[dict[int]]

indra.databases.cbio_client.**get_ccle_lines_for_mutation**(*gene*, *amino_acid_change*)
Return cell lines with a given point mutation in a given gene.

Checks which cell lines in CCLE have a particular point mutation in a given gene and return their names in a list.

Parameters

- **gene** (*str*) – The HGNC symbol of the mutated gene in whose product the amino acid change occurs. Example: "BRAF"
- **amino_acid_change** (*str*) – The amino acid change of interest. Example: "V600E"

Returns **cell_lines** – A list of CCLE cell lines in which the given mutation occurs.

Return type list

indra.databases.cbio_client.**get_ccle_mrna**(*gene_list*, *cell_lines*)
Return a dict of mRNA amounts in given genes and cell lines from CCLE.

Parameters

- **gene_list** (*list[str]*) – A list of HGNC gene symbols to get mRNA amounts for.
- **cell_lines** (*list[str]*) – A list of CCLE cell line names to get mRNA amounts for.

Returns **mrna_amounts** – A dict keyed to cell lines containing a dict keyed to genes containing float

Return type dict[dict[float]]

indra.databases.cbio_client.**get_ccle_mutations**(*gene_list*, *cell_lines*, *mutation_type=None*)
Return a dict of mutations in given genes and cell lines from CCLE.

This is a specialized call to get_mutations tailored to CCLE cell lines.

Parameters

- **gene_list** (*list[str]*) – A list of HGNC gene symbols to get mutations in
- **cell_lines** (*list[str]*) – A list of CCLE cell line names to get mutations for.
- **mutation_type** (*Optional[str]*) – The type of mutation to filter to. mutation_type can be one of: missense, nonsense, frame_shift_ins, frame_shift_del, splice_site

Returns

mutations – The result from cBioPortal as a dict in the format {cell_line : {gene : [mutation1, mutation2, ...] }}

Example: {'LOXIMVI_SKIN': {'BRAF': ['V600E', 'I208V']}, 'SKMEL30_SKIN': {'BRAF': ['D287H', 'E275K']}}

Return type dict

indra.databases.cbio_client.**get_genetic_profiles**(*study_id*, *profile_filter=None*)
Return all the genetic profiles (data sets) for a given study.

Genetic profiles are different types of data for a given study. For instance the study 'cellline_ccle_broad' has profiles such as 'cellline_ccle_broad_mutations' for mutations, 'cellline_ccle_broad_CNA' for copy number alterations, etc.

Parameters

- **study_id** (*str*) – The ID of the cBio study. Example: 'paad_icgc'
- **profile_filter** (*Optional[str]*) – A string used to filter the profiles to return. Will be one of: - MUTATION - MUTATION_EXTENDED - COPY_NUMBER_ALTERATION - MRNA_EXPRESSION - METHYLATION The genetic profiles can include "mutation", "CNA", "rppa", "methylation", etc.

Returns **genetic_profiles** – A list of genetic profiles available for the given study.

Return type list[str]

indra.databases.cbio_client.**get_mutations**(*study_id*, *gene_list*, *mutation_type=None*, *case_id=None*)
Return mutations as a list of genes and list of amino acid changes.

Parameters

- **study_id** (*str*) – The ID of the cBio study. Example: 'cellline_ccle_broad' or 'paad_icgc'
- **gene_list** (*list[str]*) – A list of genes with their HGNC symbols. Example: ['BRAF', 'KRAS']
- **mutation_type** (*Optional[str]*) – The type of mutation to filter to. mutation_type can be one of: missense, nonsense, frame_shift_ins, frame_shift_del, splice_site
- **case_id** (*Optional[str]*) – The case ID within the study to filter to.

Returns **mutations** – A tuple of two lists, the first one containing a list of genes, and the second one a list of amino acid changes in those genes.

Return type tuple[list]

indra.databases.cbio_client.**get_num_sequenced**(*study_id*)
Return number of sequenced tumors for given study.

This is useful for calculating mutation statistics in terms of the prevalence of certain mutations within a type of cancer.

Parameters **study_id** (*str*) – The ID of the cBio study. Example: 'paad_icgc'

Returns **num_case** – The number of sequenced tumors in the given study

Return type int

indra.databases.cbio_client.**get_profile_data**(*study_id*, *gene_list*, *profile_filter*, *case_set_filter=None*)
Return dict of cases and genes and their respective values.

Parameters

- **study_id** (*str*) – The ID of the cBio study. Example: 'cellline_ccle_broad' or 'paad_icgc'
- **gene_list** (*list[str]*) – A list of genes with their HGNC symbols. Example: ['BRAF', 'KRAS']
- **profile_filter** (*str*) – A string used to filter the profiles to return. Will be one of: - MUTATION - MUTATION_EXTENDED - COPY_NUMBER_ALTERATION - MRNA_EXPRESSION - METHYLATION
- **case_set_filter** (*Optional[str]*) – A string that specifices which case_set_id to use, based on a complete or partial match. If not provided, will look for study_id + '_all'

Returns **profile_data** – A dict keyed to cases containing a dict keyed to genes containing int

Return type dict[dict[int]]

indra.databases.cbio_client.**send_request**(***kwargs*)
Return a data frame from a web service request to cBio portal.

Sends a web service requrest to the cBio portal with arguments given in the dictionary data and returns a Pandas data frame on success.

More information about the service here: http://www.cbioportal.org/web_api.jsp

Parameters **kwargs** (*dict*) – A dict of parameters for the query. Entries map directly to web service calls with the exception of the optional 'skiprows' entry, whose value is used as the number of rows to skip when reading the result data frame.

Returns **df** – Response from cBioPortal as a Pandas DataFrame.

Return type pandas.DataFrame

4.3.8 ChEMBL client (indra.databases.chembl_client)

indra.databases.chembl_client.**activities_by_target**(*activities*)
Get back lists of activities in a dict keyed by ChEMBL target id

Parameters **activities** (*list*) – response from a query returning activities for a drug

Returns **targ_act_dict** – dictionary keyed to ChEMBL target ids with lists of activity ids

Return type dict

indra.databases.chembl_client.**get_chembl_id**(*nlm_mesh*)
Get ChEMBL ID from NLM MESH

Parameters **nlm_mesh** (*str*) –

Returns **chembl_id**

Return type str

indra.databases.chembl_client.**get_chembl_name**(*chembl_id*)
Return a standard ChEMBL name from an ID if available in the local resource.

Parameters **chembl_id** (*str*) – The ChEBML ID to get the name for.

Returns The corresponding ChEBML name or None if not available.

Return type str or None

indra.databases.chembl_client.**get_drug_inhibition_stmts**(*drug*)
Query ChEMBL for kinetics data given drug as Agent get back statements

Parameters **drug** (Agent) – Agent representing drug with MESH or CHEBI grounding

Returns **stmts** – INDRA statements generated by querying ChEMBL for all kinetics data of a drug interacting with protein targets

Return type list of INDRA statements

indra.databases.chembl_client.**get_evidence**(*assay*)
Given an activity, return an INDRA Evidence object.

Parameters **assay** (*dict*) – an activity from the activities list returned by a query to the API

Returns **ev** – an Evidence object containing the kinetics of the

Return type Evidence

indra.databases.chembl_client.**get_kinetics**(*assay*)
Given an activity, return its kinetics values.

Parameters **assay** (*dict*) – an activity from the activities list returned by a query to the API

Returns **kin** – dictionary of values with units keyed to value types 'IC50', 'EC50', 'INH', 'Potency', 'Kd'

Return type dict

indra.databases.chembl_client.**get_mesh_id**(*nlm_mesh*)
Get MESH ID from NLM MESH

Parameters **nlm_mesh** (*str*) –

Returns **mesh_id**

Return type str

indra.databases.chembl_client.**get_pcid**(*mesh_id*)
Get PC ID from MESH ID

Parameters **mesh** (*str*) –

Returns **pcid**

Return type str

indra.databases.chembl_client.**get_pmid**(*doc_id*)
Get PMID from document_chembl_id

Parameters **doc_id** (*str*) –

Returns **pmid**

Return type str

indra.databases.chembl_client.**get_protein_targets_only**(*target_chembl_ids*)
Given list of ChEMBL target ids, return dict of SINGLE PROTEIN targets

Parameters **target_chembl_ids** (*list*) – list of chembl_ids as strings

Returns **protein_targets** – dictionary keyed to ChEMBL target ids with lists of activity ids

Return type dict

indra.databases.chembl_client.**get_target_chemblid**(*target_upid*)
Get ChEMBL ID from UniProt upid

Parameters **target_upid** (*str*) –

Returns **target_chembl_id**

Return type str

indra.databases.chembl_client.**query_target**(*target_chembl_id*)
Query ChEMBL API target by id

Parameters **target_chembl_id** (*str*) –

Returns **target** – dict parsed from json that is unique for the target

Return type dict

indra.databases.chembl_client.**send_query**(*query_dict*)
Query ChEMBL API

Parameters **query_dict** (*dict*) – 'query' : string of the endpoint to query 'params' : dict of params for the query

Returns **js** – dict parsed from json that is unique to the submitted query

Return type dict

4.3.9 LINCS client (indra.databases.lincs_client)

class indra.databases.lincs_client.**LincsClient**
Client for querying LINCS small molecules and proteins.

get_protein_refs(*hms_lincs_id*)
Get the refs for a protein from the LINCs protein metadata.

Parameters **hms_lincs_id** (*str*) – The HMS LINCS ID for the protein

Returns A dictionary of protein references.

Return type dict

get_small_molecule_name(*hms_lincs_id*)
Get the name of a small molecule from the LINCS sm metadata.

Parameters **hms_lincs_id** (*str*) – The HMS LINCS ID of the small molecule.

Returns The name of the small molecule.

Return type str

get_small_molecule_refs(*hms_lincs_id*)
Get the id refs of a small molecule from the LINCS sm metadata.

Parameters **hms_lincs_id** (*str*) – The HMS LINCS ID of the small molecule.

Returns A dictionary of references.

Return type dict

indra.databases.lincs_client.**get_drug_target_data**()
Load the csv into a list of dicts containing the LINCS drug target data.

Returns **data** – A list of dicts, each keyed based on the header of the csv, with values as the corresponding column values.

Return type list[dict]

indra.databases.lincs_client.**load_lincs_csv**(*url*)
Helper function to turn csv rows into dicts.

4.3.10 MeSH client (indra.databases.mesh_client)

indra.databases.mesh_client.**get_db_mapping**(*mesh_id*)
Return mapping to another name space for a MeSH ID, if it exists.

Parameters **mesh_id** (*str*) – The MeSH ID whose mappings is to be returned.

Returns A tuple consisting of a DB namespace and ID for the mapping or None if not available.

Return type tuple or None

indra.databases.mesh_client.**get_go_id**(*mesh_id*)
Return a GO ID corresponding to the given MeSH ID.

Parameters **mesh_id** (*str*) – MeSH ID to map to GO

Returns The GO ID corresponding to the given MeSH ID, or None if not available.

Return type str

indra.databases.mesh_client.**get_mesh_id_from_db_id**(*db_ns*, *db_id*)
Return a MeSH ID mapped from another namespace and ID.

Parameters

- **db_ns** (*str*) – A namespace corresponding to db_id.
- **db_id** (*str*) – An ID in the given namespace.

Returns The MeSH ID corresponding to the given namespace and ID if available, otherwise None.

Return type str or None

indra.databases.mesh_client.**get_mesh_id_from_go_id**(*go_id*)
Return a MeSH ID corresponding to the given GO ID.

Parameters **go_id** (*str*) – GO ID to map to MeSH

Returns The MeSH ID corresponding to the given GO ID, or None if not available.

Return type str

indra.databases.mesh_client.**get_mesh_id_name**(*mesh_term*, *offline=False*)
Get the MESH ID and name for the given MESH term.

Uses the mappings table in *indra/resources*; if the MESH term is not listed there, falls back on the NLM REST API.

Parameters

- **mesh_term** (*str*) – MESH Descriptor or Concept name, e.g. 'Breast Cancer'.

- **offline** (*bool*) – Whether to allow queries to the NLM REST API if the given MESH term is not contained in INDRA's internal MESH mappings file. Default is False (allows REST API queries).

Returns Returns a 2-tuple of the form *(id, name)* with the ID of the descriptor corresponding to the MESH label, and the descriptor name (which may not exactly match the name provided as an argument if it is a Concept name). If the query failed, or no descriptor corresponding to the name was found, returns a tuple of (None, None).

Return type tuple of strs

indra.databases.mesh_client.**get_mesh_id_name_from_web**(*mesh_term*)
Get the MESH ID and name for the given MESH term using the NLM REST API.

Parameters **mesh_term** (*str*) – MESH Descriptor or Concept name, e.g. 'Breast Cancer'.

Returns Returns a 2-tuple of the form *(id, name)* with the ID of the descriptor corresponding to the MESH label, and the descriptor name (which may not exactly match the name provided as an argument if it is a Concept name). If the query failed, or no descriptor corresponding to the name was found, returns a tuple of (None, None).

Return type tuple of strs

indra.databases.mesh_client.**get_mesh_name**(*mesh_id*, *offline=False*)
Get the MESH label for the given MESH ID.

Uses the mappings table in *indra/resources*; if the MESH ID is not listed there, falls back on the NLM REST API.

Parameters

- **mesh_id** (*str*) – MESH Identifier, e.g. 'D003094'.
- **offline** (*bool*) – Whether to allow queries to the NLM REST API if the given MESH ID is not contained in INDRA's internal MESH mappings file. Default is False (allows REST API queries).

Returns Label for the MESH ID, or None if the query failed or no label was found.

Return type str

indra.databases.mesh_client.**get_mesh_name_from_web**(*mesh_id*)
Get the MESH label for the given MESH ID using the NLM REST API.

Parameters **mesh_id** (*str*) – MESH Identifier, e.g. 'D003094'.

Returns Label for the MESH ID, or None if the query failed or no label was found.

Return type str

indra.databases.mesh_client.**get_mesh_tree_numbers**(*mesh_id*)
Return MeSH tree IDs associated with a MeSH ID from the resource file.

This function can handle supplementary concepts by first mapping them to primary terms and then collecting all the tree numbers for the mapped primary terms.

Parameters **mesh_id** (*str*) – The MeSH ID whose tree IDs should be returned.

Returns A list of MeSH tree IDs.

Return type list[str]

indra.databases.mesh_client.**get_mesh_tree_numbers_from_web**(*mesh_id*)
Return MeSH tree IDs associated with a MeSH ID from the web.

Parameters **mesh_id** (*str*) – The MeSH ID whose tree IDs should be returned.

Returns A list of MeSH tree IDs.

Return type list[str]

indra.databases.mesh_client.**get_primary_mappings**(*db_id*)
Return the list of primary terms a supplementary term is mapped to.

See https://www.nlm.nih.gov/mesh/xml_data_elements.html#HeadingMappedTo.

Parameters **db_id** (str) – A supplementary MeSH ID.

Return type List[str]

Returns The list of primary MeSH terms that the supplementary concept is heading-mapped to.

indra.databases.mesh_client.**has_tree_prefix**(*mesh_id*, *tree_prefix*)
Return True if the given MeSH ID has the given tree prefix.

indra.databases.mesh_client.**is_disease**(*mesh_id*)
Return True if the given MeSH ID is a disease.

indra.databases.mesh_client.**is_enzyme**(*mesh_id*)
Return True if the given MeSH ID is an enzyme.

indra.databases.mesh_client.**is_molecular**(*mesh_id*)
Return True if the given MeSH ID is a chemical or drug (incl protein).

indra.databases.mesh_client.**is_protein**(*mesh_id*)
Return True if the given MeSH ID is a protein.

4.3.11 GO client (indra.databases.go_client)

A client to the Gene Ontology.

indra.databases.go_client.**get_go_id_from_label**(*label*)
Get ID corresponding to a given GO label.

Parameters **label** (*str*) – The GO label to get the ID for.

Returns Identifier corresponding to the GO label, starts with GO:.

Return type str

indra.databases.go_client.**get_go_id_from_label_or_synonym**(*label*)
Get ID corresponding to a given GO label or synonym

Parameters **label** (*str*) – The GO label or synonym to get the ID for.

Returns Identifier corresponding to the GO label or synonym, starts with GO:.

Return type str

indra.databases.go_client.**get_go_label**(*go_id*)
Get label corresponding to a given GO identifier.

Parameters **go_id** (*str*) – The GO identifier. Should include the *GO:* prefix, e.g., *GO:1903793* (positive regulation of anion transport).

Returns Label corresponding to the GO ID.

Return type str

indra.databases.go_client.**get_namespace**(*go_id*)
Return the GO namespace associated with a GO ID.

Parameters **go_id** (str) – The GO ID to get the namespace for

Return type Optional[str]

Returns The GO namespace for the given ID. This is one of molecular_function, biological_process or cellular_component. If the GO ID is not available as an entry, None is returned.

indra.databases.go_client.**get_primary_id**(*go_id*)
Get primary ID corresponding to an alternative/deprecated GO ID.

Parameters **go_id** (*str*) – The GO ID to get the primary ID for.

Returns Primary identifier corresponding to the given ID.

Return type str

indra.databases.go_client.**get_valid_location**(*loc*)
Return a valid GO label based on an ID, label or synonym.

The rationale behind this function is that many sources produce cellular locations that are arbitrarily either GO IDs (sometimes without the prefix and sometimes outdated) or labels or synonyms. This function handles all these cases and returns a valid GO label in case one is available, otherwise None.

Parameters **loc** (*txt*) – The location that needst o be canonicalized.

Returns The valid location string is available, otherwise None.

Return type str or None

4.3.12 PubChem client (indra.databases.pubchem_client)

indra.databases.pubchem_client.**get_inchi_key**(*pubchem_cid*)
Return the InChIKey for a given PubChem CID.

Parameters **pubchem_cid** (*str*) – The PubChem CID whose InChIKey should be returned.

Returns The InChIKey corresponding to the PubChem CID.

Return type str

indra.databases.pubchem_client.**get_json_record**(*pubchem_cid*)
Return the JSON record of a given PubChem CID.

Parameters **pubchem_cid** (*str*) – The PubChem CID whose record should be returned.

Returns The record deserialized from JSON.

Return type dict

indra.databases.pubchem_client.**get_mesh_id**(*pubchem_cid*)
Return the MeSH ID for a given PubChem CID.

Parameters **pubchem_cid** (str) – The PubChem CID whose MeSH ID should be returned.

Return type Optional[str]

Returns The MeSH ID corresponding to the PubChem CID or None if not available.

indra.databases.pubchem_client.**get_pmids**(*pubchem_cid*)
Return depositor provided PMIDs for a given PubChem CID.

Note that this information can also be obtained via PubMed at https://www.ncbi.nlm.nih.gov/sites/entrez?LinkName=pccompound_pubmed&db=pccompound&cmd=Link&from_uid=<pubchem_cid>.

Parameters **pubchem_cid** (str) – The PubChem CID whose PMIDs will be returned.

Return type List[str]

Returns PMIDs corresponding to the given PubChem CID. If none present, or the query fails, an empty list is returned.

indra.databases.pubchem_client.**get_preferred_compound_ids**(*pubchem_cid*)
Return a list of preferred CIDs for a given PubChem CID.

Parameters **pubchem_cid** (*str*) – The PubChem CID whose preferred CIDs should be returned.

Returns The list of preferred CIDs for the given CID. If there are no preferred CIDs for the given CID then an empty list is returned.

Return type list of str

4.3.13 miRBase client (indra.databases.mirbase_client)

A client to miRBase.

indra.databases.mirbase_client.**get_hgnc_id_from_mirbase_id**(*mirbase_id*)
Return the HGNC ID corresponding to the given miRBase ID.

Parameters **mirbase_id** (*str*) – The miRBase ID to be converted. Example: "MI0000060"

Returns **hgnc_id** – The HGNC ID corresponding to the given miRBase ID.

Return type str

indra.databases.mirbase_client.**get_mirbase_id_from_hgnc_id**(*hgnc_id*)
Return the HGNC ID corresponding to the given miRBase ID.

Parameters **hgnc_id** (*str*) – An HGNC identifier to convert to miRBase, if it is indeed an miRNA. Example: "31476"

Returns **mirbase_id** – The miRBase ID corresponding to the given HGNC ID.

Return type str

indra.databases.mirbase_client.**get_mirbase_id_from_hgnc_symbol**(*hgnc_symbol*)
Return the HGNC gene symbol corresponding to the given miRBase ID.

Parameters **hgnc_symbol** (*str*) – An HGNC gene symbol to convert to miRBase, if it is indeed an miRNA. Example: "MIR19B2"

Returns **mirbase_id** – The miRBase ID corresponding to the given HGNC gene symbol.

Return type str

indra.databases.mirbase_client.**get_mirbase_id_from_mirbase_name**(*mirbase_name*)
Return the miRBase identifier corresponding to the given miRBase name.

Parameters **mirbase_name** (*str*) – The miRBase ID to be converted. Example: "hsa-mir-19b-2"

Returns **mirbase_id** – The miRBase ID corresponding to the given miRBase name.

Return type str

indra.databases.mirbase_client.**get_mirbase_name_from_mirbase_id**(*mirbase_id*)
Return the miRBase name corresponding to the given miRBase ID.

Parameters **mirbase_id** (*str*) – The miRBase ID to be converted. Example: "MI0000060"

Returns **mirbase_name** – The miRBase name corresponding to the given miRBase ID.

Return type str

4.3.14 Experimental Factor Ontology (EFO) client (indra.databases.efo_client)

A client to EFO.

indra.databases.efo_client.**get_efo_id_from_efo_name**(*efo_name*)
Return the EFO identifier corresponding to the given EFO name.

Parameters **efo_name** (*str*) – The EFO name to be converted. Example: "gum cancer"

Returns **efo_id** – The EFO identifier corresponding to the given EFO name.

Return type str

indra.databases.efo_client.**get_efo_name_from_efo_id**(*efo_id*)
Return the EFO name corresponding to the given EFO ID.

Parameters **efo_id** (*str*) – The EFO identifier to be converted. Example: "0005557"

Returns **efo_name** – The EFO name corresponding to the given EFO identifier.

Return type str

4.3.15 Human Phenotype Ontology (HP) client (indra.databases.hp_client)

A client to HP.

indra.databases.hp_client.**get_hp_id_from_hp_name**(*hp_name*)
Return the HP identifier corresponding to the given HP name.

Parameters **hp_name** (*str*) – The HP name to be converted. Example: "Nocturia"

Returns **hp_id** – The HP identifier corresponding to the given HP name.

Return type str

indra.databases.hp_client.**get_hp_name_from_hp_id**(*hp_id*)
Return the HP name corresponding to the given HP ID.

Parameters **hp_id** (*str*) – The HP identifier to be converted. Example: "HP:0000017"

Returns **hp_name** – The HP name corresponding to the given HP identifier.

Return type str

4.3.16 Disease Ontology (DOID) client (indra.databases.doid_client)

A client to the Disease Ontology.

indra.databases.doid_client.**get_doid_id_from_doid_alt_id**(*doid_alt_id*)
Return the identifier corresponding to the given Disease Ontology alt id.

Parameters **doid_alt_id** (*str*) – The Disease Ontology alt id to be converted. Example: "DOID:267"

Returns **doid_id** – The Disease Ontology identifier corresponding to the given alt id.

Return type str

indra.databases.doid_client.**get_doid_id_from_doid_name**(*doid_name*)
Return the identifier corresponding to the given Disease Ontology name.

Parameters **doid_name** (*str*) – The Disease Ontology name to be converted. Example: "Nocturia"

Returns **doid_id** – The Disease Ontology identifier corresponding to the given name.

Return type str

indra.databases.doid_client.**get_doid_name_from_doid_id**(*doid_id*)
Return the name corresponding to the given Disease Ontology ID.

Parameters **doid_id** (*str*) – The Disease Ontology identifier to be converted. Example: "DOID:0000017"

Returns **doid_name** – The DOID name corresponding to the given DOID identifier.

Return type str

4.3.17 Infectious Disease Ontology client (indra.databases.ido_client)

A client to OWL.

indra.databases.ido_client.**get_ido_id_from_ido_name**(*ido_name*)
Return the HP identifier corresponding to the given IDO name.

Parameters **ido_name** (*str*) – The IDO name to be converted. Example: "parasite role"

Return type Optional[str]

Returns The IDO identifier corresponding to the given IDO name.

indra.databases.ido_client.**get_ido_name_from_ido_id**(*ido_id*)
Return the HP name corresponding to the given HP ID.

Parameters **ido_id** (*str*) – The IDO identifier to be converted. Example: "0000403"

Return type Optional[str]

Returns The IDO name corresponding to the given IDO identifier.

4.3.18 Taxonomy client (indra.databases.taxonomy_client)

Client to access the Entrez Taxonomy web service.

indra.databases.taxonomy_client.**get_taxonomy_id**(*name*)
Return the taxonomy ID corresponding to a taxonomy name.

Parameters **name** (*str*) – The name of the taxonomy entry. Example: "Severe acute respiratory syndrome coronavirus 2"

Returns The taxonomy ID corresponding to the given name or None if not available.

Return type str or None

4.3.19 DrugBank client (indra.databases.drugbank_client)

Client for interacting with DrugBank entries.

indra.databases.drugbank_client.**get_chebi_id**(*drugbank_id*)
Return a mapping for a DrugBank ID to CHEBI.

Parameters **drugbank_id** (*str*) – DrugBank ID to map.

Returns The ID mapped to CHEBI or None if not available.

Return type str or None

indra.databases.drugbank_client.**get_chembl_id**(*drugbank_id*)
Return a mapping for a DrugBank ID to CHEMBL.

Parameters **drugbank_id** (*str*) – DrugBank ID to map.

Returns The ID mapped to CHEMBL or None if not available.

Return type str or None

indra.databases.drugbank_client.**get_db_mapping**(*drugbank_id*, *db_ns*)
Return a mapping for a DrugBank ID to a given name space.

Parameters

- **drugbank_id** (*str*) – DrugBank ID to map.
- **db_ns** (*str*) – The database name space to map to.

Returns The ID mapped to the given name space or None if not available.

Return type str or None

indra.databases.drugbank_client.**get_drugbank_id_from_chebi_id**(*chebi_id*)
Return DrugBank ID from a CHEBI ID.

Parameters **chebi_id** (*str*) – CHEBI ID to map.

Returns The mapped DrugBank ID or None if not available.

Return type str or None

indra.databases.drugbank_client.**get_drugbank_id_from_chembl_id**(*chembl_id*)
Return DrugBank ID from a CHEMBL ID.

Parameters **chembl_id** (*str*) – CHEMBL ID to map.

Returns The mapped DrugBank ID or None if not available.

Return type str or None

indra.databases.drugbank_client.**get_drugbank_id_from_db_id**(*db_ns*, *db_id*)
Return DrugBank ID from a database name space and ID.

Parameters

- **db_ns** (*str*) – Database name space.
- **db_id** (*str*) – Database ID.

Returns The mapped DrugBank ID or None if not available.

Return type str or None

indra.databases.drugbank_client.**get_drugbank_name**(*drugbank_id*)
Return the DrugBank standard name for a given DrugBank ID.

Parameters **drugbank_id** (*str*) – DrugBank ID to get the name for

Returns The name corresponding to the given DrugBank ID or None if not available.

Return type str or None

4.3.20 OBO client (indra.databases.obo_client)

A client for OBO-sourced identifier mappings.

class indra.databases.obo_client.**OboClient**(*prefix*)
A base client for data that's been grabbed via OBO

static entries_from_graph(*obo_graph*, *prefix*, *remove_prefix=False*, *allowed_synonyms=None*, *allowed_external_ns=None*)
Return processed entries from an OBO graph.

classmethod update_resource(*directory*, *url*, *prefix*, **args*, *remove_prefix=False*, *allowed_synonyms=None*, *allowed_external_ns=None*, *force=False*)
Write the OBO information to files in the given directory.

class indra.databases.obo_client.**OntologyClient**(*prefix*)
A base client class for OBO and OWL ontologies.

get_id_from_alt_id(*db_alt_id*)
Return the canonical database id corresponding to the alt id.

Parameters **db_alt_id** (str) – The alt id to be converted.

Return type Optional[str]

Returns The ID corresponding to the given alt id.

get_id_from_name(*db_name*)
Return the database identifier corresponding to the given name.

Parameters **db_name** (str) – The name to be converted.

Return type Optional[str]

Returns The ID corresponding to the given name.

get_id_from_name_or_synonym(*txt*)
Return the database id corresponding to the given name or synonym.

Note that the way the OboClient is constructed, ambiguous synonyms are filtered out. Further, this function prioritizes names over synonyms (i.e., it first looks up the ID by name, and only if that fails, it attempts a synonym-based lookup). Overall, these mappings are guaranteed to be many-to-one.

Parameters **txt** (str) – The name or synonym to be converted.

Return type Optional[str]

Returns The ID corresponding to the given name or synonym.

get_name_from_id(*db_id*)
Return the database name corresponding to the given database ID.

Parameters **db_id** (str) – The ID to be converted.

Return type Optional[str]

Returns The name corresponding to the given ID.

get_relation(*db_id*, *rel_type*)
Return the isa relationships corresponding to a given ID.

Parameters

- **db_id** (str) – The ID whose isa relationships should be returned
- **rel_type** (str) – The type of relationships to get, e.g., is_a, part_of

Return type List[str]

Returns The IDs of the terms that are in the given relation with the given ID.

get_relations(*db_id*)
Return the isa relationships corresponding to a given ID.

Parameters **db_id** (str) – The ID whose isa relationships should be returned

Return type Mapping[str, List[str]]

Returns A dict keyed by relation type with each entry a list of IDs of the terms that are in the given relation with the given ID.

4.3.21 OWL client (indra.databases.owl_client)

A client for OWL-sourced identifier mappings.

class indra.databases.owl_client.**OwlClient**(*prefix*)
A base client for data that's been grabbed via OWL.

static entry_from_term(*term*, *prefix*, *remove_prefix=False*, *allowed_external_ns=None*)
Create a data dictionary from a Pronto term.

Return type Mapping[str, Any]

4.3.22 Biolookup client (indra.databases.biolookup_client)

A client to the Biolookup web service available at http://biolookup.io/.

indra.databases.biolookup_client.**get_name**(*db_ns*, *db_id*)
Return the name of a namespace and corresponding ID in the Biolookup web service.

Parameters

- **db_ns** (str) – The database namespace.
- **db_id** (str) – The database ID.

Return type Dict

Returns The name of the entry.

indra.databases.biolookup_client.**lookup**(*db_ns*, *db_id*)
Look up a namespace and corresponding ID in the Biolookup web service.

Parameters

- **db_ns** (str) – The database namespace.
- **db_id** (str) – The database ID.

Return type dict

Returns A dictionary containing the results of the lookup.

indra.databases.biolookup_client.**lookup_curie**(*curie*)
Look up a CURIE in the Biolookup web service.

Parameters **curie** (str) – The CURIE to look up.

Return type Dict

Returns A dictionary containing the results of the lookup.

4.4 Literature clients (indra.literature)

indra.literature.**get_full_text**(*paper_id*, *idtype*, *preferred_content_type='text/xml'*)
Return the content and the content type of an article.

This function retreives the content of an article by its PubMed ID, PubMed Central ID, or DOI. It prioritizes full text content when available and returns an abstract from PubMed as a fallback.

Parameters

- **paper_id** (*string*) – ID of the article.
- **idtype** (*'pmid', 'pmcid', or 'doi'*) – Type of the ID.
- **preferred_content_type** (*Optional[str]r*) – Preference for full-text format, if available. Can be one of 'text/xml', 'text/plain', 'application/pdf'. Default: 'text/xml'

Returns

- **content** (*str*) – The content of the article.
- **content_type** (*str*) – The content type of the article

indra.literature.**id_lookup**(*paper_id*, *idtype*)
Take an ID of type PMID, PMCID, or DOI and lookup the other IDs.

If the DOI is not found in Pubmed, try to obtain the DOI by doing a reverse-lookup of the DOI in CrossRef using article metadata.

Parameters

- **paper_id** (*str*) – ID of the article.
- **idtype** (*str*) – Type of the ID: 'pmid', 'pmcid', or 'doi

Returns **ids** – A dictionary with the following keys: pmid, pmcid and doi.

Return type dict

4.4.1 Pubmed client (indra.literature.pubmed_client)

Search and get metadata for articles in Pubmed.

indra.literature.pubmed_client.**expand_pagination**(*pages*)
Convert a page number to long form, e.g., from 456-7 to 456-457.

indra.literature.pubmed_client.**get_abstract**(*pubmed_id*, *prepend_title=True*)
Get the abstract of an article in the Pubmed database.

indra.literature.pubmed_client.**get_article_xml**(*pubmed_id*)
Get the Article subtree a single article from the Pubmed database.

Parameters **pubmed_id** (*str*) – A PubMed ID.

Returns The XML ElementTree Element that represents the Article portion of the PubMed entry.

Return type xml.etree.ElementTree.Element

indra.literature.pubmed_client.**get_full_xml**(*pubmed_id*)
Get the full XML tree of a single article from the Pubmed database.

Parameters **pubmed_id** (*str*) – A PubMed ID.

Returns The root element of the XML tree representing the PubMed entry. The root is a PubmedArticleSet with a single PubmedArticle element that contains the article metadata.

Return type xml.etree.ElementTree.Element

indra.literature.pubmed_client.**get_id_count**(*search_term*)
Get the number of citations in Pubmed for a search query.

Parameters **search_term** (*str*) – A term for which the PubMed search should be performed.

Returns The number of citations for the query, or None if the query fails.

Return type int or None

indra.literature.pubmed_client.**get_ids**(*search_term*, ***kwargs*)
Search Pubmed for paper IDs given a search term.

Search options can be passed as keyword arguments, some of which are custom keywords identified by this function, while others are passed on as parameters for the request to the PubMed web service For details on parameters that can be used in PubMed searches, see https://www.ncbi.nlm.nih.gov/books/NBK25499/#chapter4.ESearch Some useful parameters to pass are db='pmc' to search PMC instead of pubmed reldate=2 to search for papers within the last 2 days mindate='2016/03/01', maxdate='2016/03/31' to search for papers in March 2016.

PubMed, by default, limits returned PMIDs to a small number, and this number can be controlled by the "retmax" parameter. This function uses a retmax value of 100,000 by default that can be changed via the corresponding keyword argument.

Parameters

- **search_term** (*str*) – A term for which the PubMed search should be performed.
- **use_text_word** (*Optional[bool]*) – If True, the "[tw]" string is appended to the search term to constrain the search to "text words", that is words that appear as whole in relevant parts of the PubMed entry (excl. for instance the journal name or publication date) like the title and abstract. Using this option can eliminate spurious search results such as all articles published in June for a search for the "JUN" gene, or journal names that contain Acad for a search for the "ACAD" gene. See also: https://www.nlm.nih.gov/bsd/disted/pubmedtutorial/020_760.html Default : True
- **kwargs** (*kwargs*) – Additional keyword arguments to pass to the PubMed search as parameters.

indra.literature.pubmed_client.**get_ids_for_gene**(*hgnc_name*, ***kwargs*)
Get the curated set of articles for a gene in the Entrez database.

Search parameters for the Gene database query can be passed in as keyword arguments.

Parameters **hgnc_name** (*str*) – The HGNC name of the gene. This is used to obtain the HGNC ID (using the hgnc_client module) and in turn used to obtain the Entrez ID associated with the gene. Entrez is then queried for that ID.

indra.literature.pubmed_client.**get_ids_for_mesh**(*mesh_id*, *major_topic=False*, ***kwargs*)
Return PMIDs that are annotated with a given MeSH ID.

Parameters

- **mesh_id** (*str*) – The MeSH ID of a term to search for, e.g., D009101.
- **major_topic** (*bool*) – If True, only papers for which the given MeSH ID is annotated as a major topic are returned. Otherwise all annotations are considered. Default: False
- ****kwargs** – Any further PudMed search arguments that are passed to get_ids.

indra.literature.pubmed_client.**get_issns_for_journal**(*nlm_id*)
Get a list of the ISSN numbers for a journal given its NLM ID.

Information on NLM XML DTDs is available at https://www.nlm.nih.gov/databases/dtd/

indra.literature.pubmed_client.**get_mesh_annotations**(*pmid*)
Return a list of MeSH annotations for a given PubMed ID.

Parameters **pmid** (*str*) – A PubMed ID.

Returns A list of dicts that represent MeSH annotations with the following keys: "mesh" representing the MeSH ID, "text" the standrd name associated with the MeSH ID, "major_topic" a boolean flag set depending on whether the given MeSH ID is assigned as a major topic to the article, and "qualifier" which is a MeSH qualifier ID associated with the annotation, if available, otherwise None.

Return type list of dict

indra.literature.pubmed_client.**get_metadata_for_ids**(*pmid_list*, *get_issns_from_nlm=False*, *get_abstracts=False*, *prepend_title=False*)
Get article metadata for up to 200 PMIDs from the Pubmed database.

Parameters

- **pmid_list** (*list of str*) – Can contain 1-200 PMIDs.
- **get_issns_from_nlm** (*bool*) – Look up the full list of ISSN number for the journal associated with the article, which helps to match articles to CrossRef search results. Defaults to False, since it slows down performance.
- **get_abstracts** (*bool*) – Indicates whether to include the Pubmed abstract in the results.
- **prepend_title** (*bool*) – If get_abstracts is True, specifies whether the article title should be prepended to the abstract text.

Returns Dictionary indexed by PMID. Each value is a dict containing the following fields: 'doi', 'title', 'authors', 'journal_title', 'journal_abbrev', 'journal_nlm_id', 'issn_list', 'page'.

Return type dict of dicts

indra.literature.pubmed_client.**get_metadata_from_xml_tree**(*tree*, *get_issns_from_nlm=False*, *get_abstracts=False*, *prepend_title=False*, *mesh_annotations=True*)
Get metadata for an XML tree containing PubmedArticle elements.

Documentation on the XML structure can be found at:

- https://www.nlm.nih.gov/bsd/licensee/elements_descriptions.html
- https://www.nlm.nih.gov/bsd/licensee/elements_alphabetical.html

Parameters

- **tree** (*xml.etree.ElementTree*) – ElementTree containing one or more PubmedArticle elements.
- **get_issns_from_nlm** (*Optional[bool]*) – Look up the full list of ISSN number for the journal associated with the article, which helps to match articles to CrossRef search results. Defaults to False, since it slows down performance.
- **get_abstracts** (*Optional[bool]*) – Indicates whether to include the Pubmed abstract in the results. Default: False

- **prepend_title** (*Optional[bool]*) – If get_abstracts is True, specifies whether the article title should be prepended to the abstract text. Default: False
- **mesh_annotations** (*Optional[bool]*) – If True, extract mesh annotations from the pubmed entries and include in the returned data. If false, don't. Default: True

Returns Dictionary indexed by PMID. Each value is a dict containing the following fields: 'doi', 'title', 'authors', 'journal_title', 'journal_abbrev', 'journal_nlm_id', 'issn_list', 'page'.

Return type dict of dicts

indra.literature.pubmed_client.**get_substance_annotations**(*pubmed_id*)
Return substance MeSH ID for a given PubMedID.

Note that substance annotations often overlap with MeSH annotations, however, there are cases where a substance annotation is not available under MeSH annotations.

Parameters **pubmed_id** (str) – PubMedID ID whose substance MeSH ID will be returned.

Return type List[str]

Returns Substance MeSH IDs corresponding to the given PubMed paper or if None present or a failed query, an empty list will be returned.

indra.literature.pubmed_client.**get_title**(*pubmed_id*)
Get the title of an article in the Pubmed database.

4.4.2 Pubmed Central client (indra.literature.pmc_client)

indra.literature.pmc_client.**extract_paragraphs**(*xml_string*)
Returns list of paragraphs in an NLM XML.

This returns a list of the plaintexts for each paragraph and title in the input XML, excluding some paragraphs with text that should not be relevant to biomedical text processing.

Relevant text includes titles, abstracts, and the contents of many body paragraphs. Within figures, tables, and floating elements, only captions are retained (One exception is that all paragraphs within floating boxed-text elements are retained. These elements often contain short summaries enriched with useful information.) Due to captions, nested paragraphs can appear in an NLM XML document. Occasionally there are multiple levels of nesting. If nested paragraphs appear in the input document their texts are returned in a pre-ordered traversal. The text within child paragraphs is not included in the output associated to the parent. Each parent appears in the output before its children. All children of an element appear before the elements following sibling.

All tags are removed from each paragraph in the list that is returned. LaTeX surrounded by <tex-math> tags is removed entirely.

Note: Some articles contain subarticles which are processed slightly differently from the article body. Only text from the body element of a subarticle is included, and all unwanted elements are excluded along with their captions. Boxed-text elements are excluded as well.

Parameters **xml_string** (*str*) – String containing valid NLM XML.

Returns List of extracted paragraphs from the input NLM XML

Return type list of str

indra.literature.pmc_client.**extract_text**(*xml_string*)
Get plaintext from the body of the given NLM XML string.

This plaintext consists of all paragraphs returned by indra.literature.pmc_client.extract_paragraphs separated by newlines and then finally terminated by a newline. See the DocString of extract_paragraphs for more information.

Parameters **xml_string** (*str*) – String containing valid NLM XML.

Returns Extracted plaintext.

Return type str

indra.literature.pmc_client.**filter_pmids**(*pmid_list*, *source_type*)
Filter a list of PMIDs for ones with full text from PMC.

Parameters

- **pmid_list** (*list of str*) – List of PMIDs to filter.
- **source_type** (*string*) – One of 'fulltext', 'oa_xml', 'oa_txt', or 'auth_xml'.

Returns PMIDs available in the specified source/format type.

Return type list of str

indra.literature.pmc_client.**get_xml**(*pmc_id*)
Returns XML for the article corresponding to a PMC ID.

indra.literature.pmc_client.**id_lookup**(*paper_id*, *idtype=None*)
This function takes a Pubmed ID, Pubmed Central ID, or DOI and use the Pubmed ID mapping service and looks up all other IDs from one of these. The IDs are returned in a dictionary.

4.4.3 bioRxiv client (indra.literature.biorxiv_client)

A client to obtain metadata and text content from bioRxiv (and to some extent medRxiv) preprints.

indra.literature.biorxiv_client.**get_collection_dois**(*collection_id*, *min_date=None*)
Get list of DOIs from a biorxiv/medrxiv collection.

Parameters

- **collection_id** (*str*) – The identifier of the collection to fetch.
- **min_date** (*Optional[datetime.datetime]*) – A datetime object representing an cutoff. If given, only publications that were released on or after the given date are returned. By default, no date constraint is applied.

Returns The list of DOIs in the collection.

Return type list of dict

indra.literature.biorxiv_client.**get_collection_pubs**(*collection_id*, *min_date=None*)
Get list of DOIs from a biorxiv/medrxiv collection.

Parameters

- **collection_id** (*str*) – The identifier of the collection to fetch.
- **min_date** (*Optional[datetime.datetime]*) – A datetime object representing an cutoff. If given, only publications that were released on or after the given date are returned. By default, no date constraint is applied.

Returns A list of the publication entries which include the abstract and other metadata.

Return type list of dict

indra.literature.biorxiv_client.**get_content_from_pub_json**(*pub*, *format*)
Get text content based on a given format from a publication JSON.

In the case of abstract, the content is returned from the JSON directly. For pdf, the content is returned as bytes that can be dumped into a file. For txt and xml, the text is processed out of either the raw XML or text content that rxiv provides.

Parameters

- **pub** (*dict*) – The JSON dict description a publication.
- **format** (*str*) – The format, if available, via which the content should be obtained.

indra.literature.biorxiv_client.**get_formats**(*pub*)
Return formats available for a publication JSON.

Parameters **pub** (*dict*) – The JSON dict description a publication.

Returns A dict with available formats as its keys (abstract, pdf, xml, txt) and either the content (in case of abstract) or the URL (in case of pdf, xml, txt) as the value.

Return type dict

indra.literature.biorxiv_client.**get_pdf_xml_url_base**(*content*)
Return base URL to PDF/XML based on the content of the landing page.

Parameters **content** (*str*) – The content of the landing page for an rxiv paper.

Returns The base URL if available, otherwise None.

Return type str or None

indra.literature.biorxiv_client.**get_text_from_rxiv_text**(*rxiv_text*)
Return clean text from the raw rxiv text content.

This function parses out the title, headings and subheadings, and the content of sections under headings/subheadings. It filters out some irrelevant content e.g., references and footnotes.

Parameters **rxiv_text** (*str*) – The content of the rxiv full text as obtained from the web.

Returns The text content stripped out from the raw full text.

Return type str

indra.literature.biorxiv_client.**get_text_from_rxiv_xml**(*rxiv_xml*)
Return clean text from the raw rxiv xml content.

Parameters **rxiv_xml** (*str*) – The content of the rxiv full xml as obtained from the web.

Returns The text content stripped out from the raw full xml.

Return type str

indra.literature.biorxiv_client.**get_text_url_base**(*content*)
Return base URL to full text based on the content of the landing page.

Parameters **content** (*str*) – The content of the landing page for an rxiv paper.

Returns The base URL if available, otherwise None.

Return type str or None

4.4.4 CrossRef client (indra.literature.crossref_client)

indra.literature.crossref_client.**doi_query**(*pmid*, *search_limit=10*)
Get the DOI for a PMID by matching CrossRef and Pubmed metadata.

Searches CrossRef using the article title and then accepts search hits only if they have a matching journal ISSN and page number with what is obtained from the Pubmed database.

indra.literature.crossref_client.**get_fulltext_links**(*doi*)
Return a list of links to the full text of an article given its DOI. Each list entry is a dictionary with keys: - URL: the URL to the full text - content-type: e.g. text/xml or text/plain - content-version - intended-application: e.g. text-mining

indra.literature.crossref_client.**get_metadata**(*doi*)
Returns the metadata of an article given its DOI from CrossRef as a JSON dict

4.4.5 COCI client (indra.literature.coci_client)

Client to COCI, the OpenCitations Index of Crossref open DOI-to-DOI citations.

For more information on the COCI, see: https://opencitations.net/index/coci with API documentation at https://opencitations.net/index/coci/api/v1/.

indra.literature.coci_client.**get_citation_count_for_doi**(*doi*)
Return the citation count for a given DOI.

Note that the COCI API returns a count of 0 for DOIs that are not indexed.

Parameters **doi** (str) – The DOI to get the citation count for.

Return type int

Returns The citation count for the DOI.

indra.literature.coci_client.**get_citation_count_for_pmid**(*pmid*)
Return the citation count for a given PMID.

This uses the CrossRef API to get the DOI for the PMID, and then calls the COCI API to get the citation count for the DOI.

If the DOI lookup failed, this returns None. Note that the COCI API returns a count of 0 for DOIs that are not indexed.

Parameters **pmid** (str) – The PMID to get the citation count for.

Return type Optional[int]

Returns The citation count for the PMID.

4.4.6 Elsevier client (indra.literature.elsevier_client)

For information on the Elsevier API, see:

- API Specification: http://dev.elsevier.com/api_docs.html
- Authentication: https://dev.elsevier.com/tecdoc_api_authentication.html

indra.literature.elsevier_client.**check_entitlement**(*doi*)
Check whether IP and credentials enable access to content for a doi.

This function uses the entitlement endpoint of the Elsevier API to check whether an article is available to a given institution. Note that this feature of the API is itself not available for all institution keys.

indra.literature.elsevier_client.**download_article**(*id_val*, *id_type='doi'*, *on_retry=False*)
Low level function to get an XML article for a particular id.

Parameters

- **id_val** (*str*) – The value of the id.
- **id_type** (*str*) – The type of id, such as pmid (a.k.a. pubmed_id), doi, or eid.
- **on_retry** (*bool*) – This function has a recursive retry feature, and this is the only time this parameter should be used.

Returns **content** – If found, the content string is returned, otherwise, None is returned.

Return type str or None

indra.literature.elsevier_client.**download_article_from_ids**(***id_dict*)
Download an article in XML format from Elsevier matching the set of ids.

Parameters **<id_type>** (*str*) – You can enter any combination of eid, doi, pmid, and/or pii. Ids will be checked in that order, until either content has been found or all ids have been checked.

Returns **content** – If found, the content is returned as a string, otherwise None is returned.

Return type str or None

indra.literature.elsevier_client.**download_from_search**(*query_str*, *folder*, *do_extract_text=True*, *max_results=None*)
Save raw text files based on a search for papers on ScienceDirect.

This performs a search to get PIIs, downloads the XML corresponding to the PII, extracts the raw text and then saves the text into a file in the designated folder.

Parameters

- **query_str** (*str*) – The query string to search with
- **folder** (*str*) – The local path to an existing folder in which the text files will be dumped
- **do_extract_text** (*bool*) – Choose whether to extract text from the xml, or simply save the raw xml files. Default is True, so text is extracted.
- **max_results** (*int or None*) – Default is None. If specified, limit the number of results to the given maximum.

indra.literature.elsevier_client.**extract_paragraphs**(*xml_string*)
Get paragraphs from the body of the given Elsevier xml.

indra.literature.elsevier_client.**extract_text**(*xml_string*)
Get text from the body of the given Elsevier xml.

indra.literature.elsevier_client.**get_abstract**(*doi*)
Get the abstract text of an article from Elsevier given a doi.

indra.literature.elsevier_client.**get_article**(*doi*, *output_format='txt'*)
Get the full body of an article from Elsevier.

Parameters

- **doi** (*str*) – The doi for the desired article.
- **output_format** (*'txt' or 'xml'*) – The desired format for the output. Selecting 'txt' (default) strips all xml tags and joins the pieces of text in the main text, while 'xml' simply takes the tag containing the body of the article and returns it as is . In the latter case, downstream code needs to be able to interpret Elsever's XML format.

Returns **content** – Either text content or xml, as described above, for the given doi.

Return type str

indra.literature.elsevier_client.**get_dois**(*query_str*, *year=None*, *loaded_after=None*)
Search ScienceDirect through the API for articles and return DOIs.

Parameters

- **query_str** (*str*) – The query string to search with.
- **year** (*Optional[str]*) – The year to constrain the search to.
- **loaded_after** (*Optional[str]*) – Date formatted as 'yyyy-MM-dd'T'HH:mm:ssX' to constrain the search to articles loaded after this date. Example: 2019-06-01T00:00:00Z

Returns **dois** – The list of DOIs identifying the papers returned by the search.

Return type list[str]

indra.literature.elsevier_client.**get_piis**(*query_str*)
Search ScienceDirect through the API for articles and return PIIs.

Note that ScienceDirect has a limitation in which a maximum of 6,000 PIIs can be retrieved for a given search and therefore this call is internally broken up into multiple queries by a range of years and the results are combined.

Parameters **query_str** (*str*) – The query string to search with

Returns **piis** – The list of PIIs identifying the papers returned by the search

Return type list[str]

indra.literature.elsevier_client.**get_piis_for_date**(*query_str*, *year=None*, *loaded_after=None*)
Search ScienceDirect through the API for articles and return PIIs.

Parameters

- **query_str** (*str*) – The query string to search with.
- **year** (*Optional[str]*) – The year to constrain the search to.
- **loaded_after** (*Optional[str]*) – Date formatted as 'yyyy-MM-dd'T'HH:mm:ssX' to constrain the search to articles loaded after this date. Example: 2019-06-01T00:00:00Z

Returns **piis** – The list of PIIs identifying the papers returned by the search.

Return type list[str]

indra.literature.elsevier_client.**search_science_direct**(*query_str*, *field_name*, *year=None*, *loaded_after=None*)
Search ScienceDirect for a given field with a query string.

Users can specify which field they are interested in and only values from that field will be returned. It is also possible to restrict the search either to a specific year of publication or to papers published after a specific date.

Parameters

- **query_str** (*str*) – The query string to search with.
- **field_name** (*str*) – A name of the field of interest to be returned. Accepted values are: authors, doi, loadDate, openAccess, pages, pii, publicationDate, sourceTitle, title, uri, volumeIssue.
- **year** (*Optional[str]*) – The year to constrain the search to.
- **loaded_after** (*Optional[str]*) – Date formatted as 'yyyy-MM-dd'T'HH:mm:ssX' to constrain the search to articles loaded after this date.

Returns **all_parts** – The list of values from the field of interest identifying the papers returned by the search.

Return type list[str]

4.4.7 NewsAPI client (indra.literature.newsapi_client)

This module provides a client for the NewsAPI web service (https://newsapi.org/). The web service requires an API key which is available after registering at https://newsapi.org/account. This key can be set as NEWSAPI_API_KEY in the INDRA config file or as an environmental variable with the same name.

NewsAPI also requires attribution e.g. "powered by NewsAPI.org" for derived uses.

indra.literature.newsapi_client.**send_request**(*endpoint*, ***kwargs*)

Return the response to a query as JSON from the NewsAPI web service.

The basic API is limited to 100 results which is chosen unless explicitly given as an argument. Beyond that, paging is supported through the "page" argument, if needed.

Parameters

- **endpoint** (*str*) – Endpoint to query, e.g. "everything" or "top-headlines"
- **kwargs** (*dict*) – A list of keyword arguments passed as parameters with the query. The basic ones are "q" which is the search query, "from" is a start date formatted as for instance 2018-06-10 and "to" is an end date with the same format.

Returns **res_json** – The response from the web service as a JSON dict.

Return type dict

4.4.8 Adeft Tools (indra.literature.adeft_tools)

This file provides several functions helpful for acquiring texts for Adeft disambiguation.

It offers the ability to get text content for articles containing a particular gene. This is useful for aquiring training texts for genes genes that do not appear in a defining pattern with a problematic shortform.

General XML processing is also provided that allows for extracting text from a source that may be either of Elsevier XML, NLM XML or raw text. This is helpful because it avoids having to know in advance the source of text content from the database.

indra.literature.adeft_tools.**filter_paragraphs**(*paragraphs*, *contains=None*)

Filter paragraphs to only those containing one of a list of strings

Parameters

- **paragraphs** (*list of str*) – List of plaintext paragraphs from an article
- **contains** (*str or list of str*) – Exclude paragraphs not containing this string as a token, or at least one of the strings in contains if it is a list

Returns Plaintext consisting of all input paragraphs containing at least one of the supplied tokens.

Return type str

indra.literature.adeft_tools.**get_text_content_for_gene**(*hgnc_name*)

Get articles that have been annotated to contain gene in entrez

Parameters **hgnc_name** (*str*) – HGNC name for gene

Returns **text_content** – xmls of fulltext if available otherwise abstracts for all articles that haven been annotated in entrez to contain the given gene

Return type list of str

indra.literature.adeft_tools.**get_text_content_for_pmids**(*pmids*)
Get text content for articles given a list of their pmids

Parameters **pmids** (*list of str*) –

Returns **text_content**

Return type list of str

indra.literature.adeft_tools.**universal_extract_paragraphs**(*xml*)
Extract paragraphs from xml that could be from different sources

First try to parse the xml as if it came from elsevier. if we do not have valid elsevier xml this will throw an exception. the text extraction function in the pmc client may not throw an exception when parsing elsevier xml, silently processing the xml incorrectly

Parameters **xml** (*str*) – Either an NLM xml, Elsevier xml or plaintext

Returns **paragraphs** – Extracted plaintext paragraphs from NLM or Elsevier XML

Return type str

indra.literature.adeft_tools.**universal_extract_text**(*xml*, *contains=None*)
Extract plaintext from xml that could be from different sources

Parameters

- **xml** (*str*) – Either an NLM xml, Elsevier xml, or plaintext
- **contains** (*str or list of str*) – Exclude paragraphs not containing this string, or at least one of the strings in contains if it is a list

Returns The concatentation of all paragraphs in the input xml, excluding paragraphs not containing one of the tokens in the list contains. Paragraphs are separated by new lines.

Return type str

4.5 INDRA Ontologies (indra.ontology)

4.5.1 IndraOntology (indra.ontology)

class indra.ontology.ontology_graph.**IndraOntology**
A directed graph representing entities and their properties as nodes and ontological relationships between the entities as edges.

name
A prefix/name for the ontology, used for the purposes of caching.

Type str

version
A version for the ontology, used for the purposes of caching.

Type str

get_children(*ns*, *id*, *ns_filter=None*)
Return all *isa* or *partof* children of a given entity.

Importantly, *isa* and *partof* edges always point towards higher-level entities in the ontology but here "child" means lower-level entity i.e., ancestors in the graph.

Parameters

- **ns** (*str*) – The name space of an entity.
- **id** (*str*) – The ID of an entity.
- **ns_filter** (*Optional[set]*) – If provided, only entities within the set of given name spaces are returned.

Returns A list of entities (name space, ID pairs) that are the children of the given entity.

Return type list

static get_id(*node*)
Return the name ID a given node from its label.

Parameters **node** (*str*) – A node's label.

Returns The node's ID within its name space.

Return type str

get_id_from_name(*ns*, *name*)
Return an entity's ID given its name space and standard name.

Parameters

- **ns** (*str*) – The name space in which the standard name is defined.
- **name** (*str*) – The standard name defined in the name space.

Return type Optional[Tuple[str, str]]

Returns The pair of namespace and ID corresponding to the given standard name in the given name space or None if it's not available.

get_mappings(*ns*, *id*)
Return entities that are xrefs of a given entity.

This function returns all mappings via *xrefs* edges from the given entity.

Parameters

- **ns** (*str*) – An entity's name space.
- **id** (*str*) – An entity's ID.

Returns A list of entities (name space, ID pairs) that are direct or indirect xrefs of the given entity.

Return type list

get_name(*ns*, *id*)
Return the standard name of a given entity.

Parameters

- **ns** (*str*) – An entity's name space.
- **id** (*str*) – An entity's ID.

Returns The name associated with the given entity or None if the node is not in the ontology or doesn't have a standard name.

Return type str or None

`get_node_property`(*ns*, *id*, *property*)
Return a given property of a given entity.

Parameters

- **ns** (*str*) – An entity's name space.
- **id** (*str*) – An entity's ID.
- **property** (*str*) – The property to look for on the given node.

Returns The name associated with the given entity or None if the node is not in the ontology or doesn't have the given property.

Return type str or None

`static get_ns`(*node*)
Return the name space of a given node from its label.

Parameters **node** (*str*) – A node's label.

Returns The node's name space.

Return type str

`static get_ns_id`(*node*)
Return the name space and ID of a given node from its label.

Parameters **node** (*str*) – A node's label.

Returns A tuple of the node's name space and ID.

Return type tuple(str, str)

`get_parents`(*ns*, *id*)
Return all *isa* or *partof* parents of a given entity.

Importantly, *isa* and *partof* edges always point towards higher-level entities in the ontology but here "parent" means higher-level entity i.e., descendants in the graph.

Parameters

- **ns** (*str*) – The name space of an entity.
- **id** (*str*) – The ID of an entity.

Returns A list of entities (name space, ID pairs) that are the parents of the given entity.

Return type list

`get_polarity`(*ns*, *id*)
Return the polarity of a given entity.

Parameters

- **ns** (*str*) – An entity's name space.
- **id** (*str*) – An entity's ID.

Returns The polarity associated with the given entity or None if the node is not in the ontology or doesn't have a polarity.

Return type str or None

get_top_level_parents(*ns*, *id*)
Return all top-level *isa* or *partof* parents of a given entity.

Top level means that this function only returns parents which don't have any further *isa* or *partof* parents above them. Importantly, *isa* and *partof* edges always point towards higher-level entities in the ontology but here "parent" means higher-level entity i.e., descendants in the graph.

Parameters

- **ns** (*str*) – The name space of an entity.
- **id** (*str*) – The ID of an entity.

Returns A list of entities (name space, ID pairs) that are the top-level parents of the given entity.

Return type list

get_type(*ns*, *id*)
Return the type of a given entity.

Parameters

- **ns** (*str*) – An entity's name space.
- **id** (*str*) – An entity's ID.

Returns The type associated with the given entity or None if the node is not in the ontology or doesn't have a type annotation.

Return type str or None

initialize()
Initialize the ontology by adding nodes and edges.

By convention, ontologies are implemented such that the constructor does not add all the nodes and edges, which can take a long time. This function is called automatically when any of the user-facing methods ot IndraOntology is called. This way, the ontology is only fully constructed if it is used.

is_opposite(*ns1*, *id1*, *ns2*, *id2*)
Return True if the two entities are opposites of each other.

Parameters

- **ns1** (*str*) – The first entity's name space.
- **id1** (*str*) – The first entity's ID.
- **ns2** (*str*) – The second entity's name space.
- **id2** (*str*) – The second entity's ID.

Returns True if the first entity is in an *is_opposite* relationship with the second. False otherwise.

Return type bool

isa(*ns1*, *id1*, *ns2*, *id2*)
Return True if the first entity is related to the second as 'isa'.

Parameters

- **ns1** (*str*) – The first entity's name space.
- **id1** (*str*) – The first entity's ID.
- **ns2** (*str*) – The second entity's name space.
- **id2** (*str*) – The second entity's ID.

Returns True if the first entity is related to the second with a directed path containing edges with type *isa*. Otherwise False.

Return type bool

`isa_or_partof`(*ns1*, *id1*, *ns2*, *id2*)
Return True if the first entity is related to the second as 'isa' or *partof*.

Parameters

- **ns1** (*str*) – The first entity's name space.
- **id1** (*str*) – The first entity's ID.
- **ns2** (*str*) – The second entity's name space.
- **id2** (*str*) – The second entity's ID.

Returns True if the first entity is related to the second with a directed path containing edges with type *isa* or *partof*. Otherwise False.

Return type bool

`isrel`(*ns1*, *id1*, *ns2*, *id2*, *rels*)
Return True if the two entities are related with a given rel.

Parameters

- **ns1** (*str*) – The first entity's name space.
- **id1** (*str*) – The first entity's ID.
- **ns2** (*str*) – The second entity's name space.
- **id2** (*str*) – The second entity's ID.
- **rels** (*iterable of str*) – A set of edge types to traverse when determining if the first entity is related to the second entity.

Returns True if the first entity is related to the second with a directed path containing edges with types in *rels* . Otherwise False.

Return type bool

`static label`(*ns*, *id*)
Return the label corresponding to a given entity.

This is mostly useful for constructing the ontology or when adding new nodes/edges. It can be overriden in subclasses to change the default mapping from ns / id to a label.

Parameters

- **ns** (*str*) – An entity's name space.
- **id** (*str*) – An entity's ID.

Returns The label corresponding to the given entity.

Return type str

`map_to`(*ns1*, *id1*, *ns2*)
Return an entity that is a unique xref of an entity in a given name space.

This function first finds all mappings via *xrefs* edges from the given first entity to the given second name space. If exactly one such mapping target is found, the target is returned. Otherwise, None is returned.

Parameters

- **ns1** (*str*) – The first entity's name space.
- **id1** (*str*) – The first entity's ID.
- **ns2** (*str*) – The second entity's name space.

Returns

- *str* – The name space of the second entity
- *str* – The ID of the second entity in the given name space.

maps_to(*ns1*, *id1*, *ns2*, *id2*)
Return True if the first entity has an xref to the second.

Parameters

- **ns1** (*str*) – The first entity's name space.
- **id1** (*str*) – The first entity's ID.
- **ns2** (*str*) – The second entity's name space.
- **id2** (*str*) – The second entity's ID.

Returns True if the first entity is related to the second with a directed path containing edges with type *xref*. Otherwise False.

Return type bool

name = None

nodes_from_suffix(*suffix*)
Return all node labels which have a given suffix.

This is useful for finding entities in ontologies where the IDs consist of paths like a/b/c/...

Parameters **suffix** (*str*) – A label suffix.

Returns A list of node labels that have the given suffix.

Return type list

partof(*ns1*, *id1*, *ns2*, *id2*)
Return True if the first entity is related to the second as 'partof'.

Parameters

- **ns1** (*str*) – The first entity's name space.
- **id1** (*str*) – The first entity's ID.
- **ns2** (*str*) – The second entity's name space.
- **id2** (*str*) – The second entity's ID.

Returns True if the first entity is related to the second with a directed path containing edges with type *partof*. Otherwise False.

Return type bool

static reverse_label(*label*)
Return the name space and ID from a given label.

This is the complement of the *label* method which reverses a label into a name space and ID.

Parameters **label** – A node label.

Returns

- *str* – The name space corresponding to the label.
- *str* – The ID corresponding to the label.

4.5.2 Grounding and name standardization (`indra.ontology.standardize`)

`indra.ontology.standardize.`**`get_standard_agent`**(*name*, *db_refs*, *ontology=None*, *ns_order=None*, ***kwargs*)

Get a standard agent based on the name, db_refs, and a any other kwargs.

name [str] The name of the agent that may not be standardized.

db_refs [dict] A dict of db refs that may not be standardized, i.e., may be missing an available UP ID corresponding to an existing HGNC ID.

ontology [Optional[indra.ontology.IndraOntology]] An IndraOntology object, if not provided, the default BioOntology is used.

ns_order [Optional[list]] A list of namespaces which are in order of priority with higher priority namespaces appearing earlier in the list.

kwargs : Keyword arguments to pass to `Agent.__init__()`.

Returns A standard agent

Return type *Agent*

`indra.ontology.standardize.`**`get_standard_name`**(*db_refs*, *ontology=None*, *ns_order=None*)

Return a standardized name for a given db refs dict.

Parameters

- **db_refs** (*dict*) – A dict of db refs that may not be standardized, i.e., may be missing an available UP ID corresponding to an existing HGNC ID.
- **ontology** (*Optional[indra.ontology.IndraOntology]*) – An IndraOntology object, if not provided, the default BioOntology is used.
- **ns_order** (*Optional[list]*) – A list of namespaces which are in order of priority with higher priority namespaces appearing earlier in the list.

Returns The standard name based on the db refs, None if not available.

Return type str or None

`indra.ontology.standardize.`**`standardize_agent_name`**(*agent*, *standardize_refs=True*, *ontology=None*, *ns_order=None*)

Standardize the name of an Agent based on grounding information.

The priority of which namespace is used as the bases for the standard name depends on

Parameters

- **agent** (*indra.statements.Agent*) – An INDRA Agent whose name attribute should be standardized based on grounding information.
- **standardize_refs** (*Optional[bool]*) – If True, this function assumes that the Agent's db_refs need to be standardized, e.g., HGNC mapped to UP. Default: True
- **ontology** (*Optional[indra.ontology.IndraOntology]*) – An IndraOntology object, if not provided, the default BioOntology is used.

- **ns_order** (*Optional[list]*) – A list of namespaces which are in order of priority with higher priority namespaces appearing earlier in the list.

Returns True if a new name was set, False otherwise.

Return type bool

indra.ontology.standardize.**standardize_db_refs**(*db_refs*, *ontology=None*, *ns_order=None*)
Return a standardized db refs dict for a given db refs dict.

Parameters

- **db_refs** (*dict*) – A dict of db refs that may not be standardized, i.e., may be missing an available UP ID corresponding to an existing HGNC ID.
- **ontology** (*Optional[indra.ontology.IndraOntology]*) – An IndraOntology object, if not provided, the default BioOntology is used.
- **ns_order** (*Optional[list]*) – A list of namespaces which are in order of priority with higher priority namespaces appearing earlier in the list.

Returns The db_refs dict with standardized entries.

Return type dict

indra.ontology.standardize.**standardize_name_db_refs**(*db_refs*, *ontology=None*, *ns_order=None*)
Return a standardized name and db refs dict for a given db refs dict.

Parameters

- **db_refs** (*dict*) – A dict of db refs that may not be standardized, i.e., may be missing an available UP ID corresponding to an existing HGNC ID.
- **ontology** (*Optional[indra.ontology.IndraOntology]*) – An IndraOntology object, if not provided, the default BioOntology is used.
- **ns_order** (*Optional[list]*) – A list of namespaces which are in order of priority with higher priority namespaces appearing earlier in the list.

Returns

- *str or None* – The standard name based on the db refs, None if not available.
- *dict* – The db_refs dict with standardized entries.

4.5.3 INDRA BioOntology (indra.ontology.bio_ontology)

Module containing the implementation of an IndraOntology for the general biology use case.

class indra.ontology.bio.**BioOntology**
Represents the ontology used for biology applications.

add_edges_from(*ebunch_to_add*, ***attr*)
Add all the edges in ebunch_to_add.

Parameters

- **ebunch_to_add** (*container of edges*) – Each edge given in the container will be added to the graph. The edges must be given as 2-tuples (u, v) or 3-tuples (u, v, d) where d is a dictionary containing edge data.
- **attr** (*keyword arguments, optional*) – Edge data (or labels or objects) can be assigned using keyword arguments.

See also:

add_edge add a single edge

add_weighted_edges_from convenient way to add weighted edges

Notes

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added.

Edge attributes specified in an ebunch take precedence over attributes specified via keyword arguments.

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edges_from([(0, 1), (1, 2)])  # using a list of edge tuples
>>> e = zip(range(0, 3), range(1, 4))
>>> G.add_edges_from(e)  # Add the path graph 0-1-2-3
```

Associate data to edges

```
>>> G.add_edges_from([(1, 2), (2, 3)], weight=3)
>>> G.add_edges_from([(3, 4), (1, 4)], label="WN2898")
```

add_nodes_from(*nodes_for_adding*, ***attr*)

Add multiple nodes.

Parameters

- **nodes_for_adding** (*iterable container*) – A container of nodes (list, dict, set, etc.). OR A container of (node, attribute dict) tuples. Node attributes are updated using the attribute dict.
- **attr** (*keyword arguments, optional (default= no attributes)*) – Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified via keyword arguments.

See also:

add_node

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_nodes_from("Hello")
>>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes(), key=str)
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1, 2], size=10)
>>> G.add_nodes_from([3, 4], weight=0.4)
```

Use (node, attrdict) tuples to update attributes for specific nodes.

```
>>> G.add_nodes_from([(1, dict(size=11)), (2, {"color": "blue"})])
>>> G.nodes[1]["size"]
11
>>> H = nx.Graph()
>>> H.add_nodes_from(G.nodes(data=True))
>>> H.nodes[1]["size"]
11
```

initialize(*rebuild=False*)

Initialize the ontology by adding nodes and edges.

By convention, ontologies are implemented such that the constructor does not add all the nodes and edges, which can take a long time. This function is called automatically when any of the user-facing methods ot IndraOntology is called. This way, the ontology is only fully constructed if it is used.

class indra.ontology.bio.ontology.**BioOntology**

Represents the ontology used for biology applications.

add_edges_from(*ebunch_to_add*, ***attr*)

Add all the edges in ebunch_to_add.

Parameters

- **ebunch_to_add** (*container of edges*) – Each edge given in the container will be added to the graph. The edges must be given as 2-tuples (u, v) or 3-tuples (u, v, d) where d is a dictionary containing edge data.
- **attr** (*keyword arguments, optional*) – Edge data (or labels or objects) can be assigned using keyword arguments.

See also:

add_edge add a single edge

add_weighted_edges_from convenient way to add weighted edges

Notes

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added.

Edge attributes specified in an ebunch take precedence over attributes specified via keyword arguments.

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edges_from([(0, 1), (1, 2)])  # using a list of edge tuples
>>> e = zip(range(0, 3), range(1, 4))
>>> G.add_edges_from(e)  # Add the path graph 0-1-2-3
```

Associate data to edges

```
>>> G.add_edges_from([(1, 2), (2, 3)], weight=3)
>>> G.add_edges_from([(3, 4), (1, 4)], label="WN2898")
```

add_nodes_from(*nodes_for_adding*, ***attr*)
Add multiple nodes.

Parameters

- **nodes_for_adding** (*iterable container*) – A container of nodes (list, dict, set, etc.). OR A container of (node, attribute dict) tuples. Node attributes are updated using the attribute dict.
- **attr** (*keyword arguments, optional (default= no attributes)*) – Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified via keyword arguments.

See also:

add_node

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_nodes_from("Hello")
>>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes(), key=str)
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1, 2], size=10)
>>> G.add_nodes_from([3, 4], weight=0.4)
```

Use (node, attrdict) tuples to update attributes for specific nodes.

```
>>> G.add_nodes_from([(1, dict(size=11)), (2, {"color": "blue"})])
>>> G.nodes[1]["size"]
11
>>> H = nx.Graph()
>>> H.add_nodes_from(G.nodes(data=True))
>>> H.nodes[1]["size"]
11
```

initialize(*rebuild=False*)
Initialize the ontology by adding nodes and edges.

By convention, ontologies are implemented such that the constructor does not add all the nodes and edges, which can take a long time. This function is called automatically when any of the user-facing methods ot IndraOntology is called. This way, the ontology is only fully constructed if it is used.

Generating and caching the BioOntology

The BioOntology is built and cached automatically during runtime. If a cached version already exists, it is loaded from the cache.

To control the build and clean up caches if necessary, one can call

```
python -m indra.ontology.bio <operation>
```

to build or clean up the INDRA bio ontology. The script takes a single operation argument which can be as follows:

- *build*: build the ontology and cache it
- *clean*: delete the current version of the ontology from the cache
- *clean-old*: delete all versions of the ontology except the current one
- *clean-all*: delete all versions of the bio ontology from the cache

4.5.4 Virtual Ontology (indra.ontology.virtual_ontology)

This module implements a virtual ontology which communicates with a REST service to perform all ontology functions.

class indra.ontology.virtual.ontology.**VirtualOntology**(*url*, *ontology='bio'*)

A virtual ontology class which uses a remote REST service to perform all operations. It is particularly useful if the host machine has limited resources and keeping the ontology graph in memory is not desirable.

Parameters

- **url** (*str*) – The base URL of the ontology graph web service.
- **ontology** (*Optional[str]*) – The identifier of the ontology recognized by the web service. Default: bio

get_id_from_name(*ns*, *name*)

Return an entity's ID given its name space and standard name.

Parameters

- **ns** (*str*) – The name space in which the standard name is defined.
- **name** (*str*) – The standard name defined in the name space.

Returns The pair of namespace and ID corresponding to the given standard name in the given name space or None if it's not available.

get_node_property(*ns*, *id*, *property*)

Return a given property of a given entity.

Parameters

- **ns** (*str*) – An entity's name space.
- **id** (*str*) – An entity's ID.
- **property** (*str*) – The property to look for on the given node.

Returns The name associated with the given entity or None if the node is not in the ontology or doesn't have the given property.

Return type str or None

initialize()
Initialize the ontology by adding nodes and edges.

By convention, ontologies are implemented such that the constructor does not add all the nodes and edges, which can take a long time. This function is called automatically when any of the user-facing methods ot IndraOntology is called. This way, the ontology is only fully constructed if it is used.

4.5.5 Ontology web service (indra.ontology.app)

This module implements IndraOntology functionalities as a web service. If instantiating an ontology directly is not desirable (for instance because of memory constraints), this app can be started on a suitable server, and an instance of the VirtualOntology class can be used to communicate with it transparently.

To start the server, run

```
python -m indra.ontology.app.app
```

or use a WSGI application server such as gunicorn (the service uses port 8002 by default, this can be changed using the *–port* argument).

Once the service is started, one option is to create an instance of *VirtualOntology(url=<service url>)* and use it as an argument in various function calls.

Another option is to set the value *INDRA_ONTOLOGY_URL=<service url>* either as an environmental variable or in the INDRA configuration file. If this value is set, INDRA will use an appropriate instance of a VirtualOntology which communicates with the service in place of the BioOntology.

4.6 Preassembly (indra.preassembler)

4.6.1 Preassembler (indra.preassembler)

class indra.preassembler.**Preassembler**(*ontology*, *stmts=None*, *matches_fun=None*, *refinement_fun=None*)
De-duplicates statements and arranges them in a specificity hierarchy.

Parameters

- **ontology** (indra.ontology.IndraOntology) – An INDRA Ontology object.
- **stmts** (list of indra.statements.Statement or None) – A set of statements to perform pre-assembly on. If None, statements should be added using the *add_statements()* method.
- **matches_fun** (*Optional[function]*) – A functon which takes a Statement object as argument and returns a string key that is used for duplicate recognition. If supplied, it overrides the use of the built-in matches_key method of each Statement being assembled.
- **refinement_fun** (*Optional[function]*) – A function which takes two Statement objects and an ontology as an argument and returns True or False. If supplied, it overrides the built-in refinement_of method of each Statement being assembled.

stmts
Starting set of statements for preassembly.

Type list of indra.statements.Statement

unique_stmts
Statements resulting from combining duplicates.

Type list of `indra.statements.Statement`

related_stmts
Top-level statements after building the refinement hierarchy.

Type list of `indra.statements.Statement`

ontology
An INDRA Ontology object.

Type dict[`indra.preassembler.ontology_graph.IndraOntology`]

add_statements(*stmts*)
Add to the current list of statements.

Parameters **stmts** (list of `indra.statements.Statement`) – Statements to add to the current list.

combine_duplicate_stmts(*stmts*)
Combine evidence from duplicate Statements.

Statements are deemed to be duplicates if they have the same key returned by the *matches_key()* method of the Statement class. This generally means that statements must be identical in terms of their arguments and can differ only in their associated *Evidence* objects.

This function keeps the first instance of each set of duplicate statements and merges the lists of Evidence from all of the other statements.

Parameters **stmts** (list of `indra.statements.Statement`) – Set of statements to de-duplicate.

Returns Unique statements with accumulated evidence across duplicates.

Return type list of `indra.statements.Statement`

Examples

De-duplicate and combine evidence for two statements differing only in their evidence lists:

```
>>> from indra.ontology.bio import bio_ontology
>>> map2k1 = Agent('MAP2K1')
>>> mapk1 = Agent('MAPK1')
>>> stmt1 = Phosphorylation(map2k1, mapk1, 'T', '185',
... evidence=[Evidence(text='evidence 1')])
>>> stmt2 = Phosphorylation(map2k1, mapk1, 'T', '185',
... evidence=[Evidence(text='evidence 2')])
>>> pa = Preassembler(bio_ontology)
>>> uniq_stmts = pa.combine_duplicate_stmts([stmt1, stmt2])
>>> uniq_stmts
[Phosphorylation(MAP2K1(), MAPK1(), T, 185)]
>>> sorted([e.text for e in uniq_stmts[0].evidence])
['evidence 1', 'evidence 2']
```

combine_duplicates()
Combine duplicates among *stmts* and save result in *unique_stmts*.

A wrapper around the method *combine_duplicate_stmts()*.

combine_related(*return_toplevel=True*, *filters=None*, ***kwargs*)
Connect related statements based on their refinement relationships.

This function takes as a starting point the unique statements (with duplicates removed) and returns a modified flat list of statements containing only those statements which do not represent a refinement of other existing statements. In other words, the more general versions of a given statement do not appear at the top level, but instead are listed in the *supports* field of the top-level statements.

If *unique_stmts* has not been initialized with the de-duplicated statements, *combine_duplicates()* is called internally.

After this function is called the attribute *related_stmts* is set as a side-effect.

The procedure for combining statements in this way involves a series of steps:

1. The statements are subjected to (built-in or user-supplied) filters that group them based on potential refinement relationships. For instance, the ontology-based filter positions each statement, based on its agent arguments, with the ontology, and determines potential refinements based on paths in the ontology graph.
2. Each statement is then compared with the set of statements it can potentially refine, as determined by the pre-filters. If the statement represents a refinement of the other (as defined by the *refinement_of()* method implemented for the Statement), then the more refined statement is added to the *supports* field of the more general statement, and the more general statement is added to the *supported_by* field of the more refined statement.
3. A new flat list of statements is created that contains only those statements that have no *supports* entries (statements containing such entries are not eliminated, because they will be retrievable from the *supported_by* fields of other statements). This list is returned to the caller.

Note: Subfamily relationships must be consistent across arguments

For now, we require that merges can only occur if the *isa* relationships are all in the *same direction for all the agents* in a Statement. For example, the two statement groups: *RAF_family -> MEK1* and *BRAF -> MEK_family* would not be merged, since BRAF *isa* RAF_family, but MEK_family is not a MEK1. In the future this restriction could be revisited.

Parameters

- **return_toplevel** (*Optional[bool]*) – If True only the top level statements are returned. If False, all statements are returned. Default: True
- **filters** (Optional[list[*indra.preassembler.refinement.RefinementFilter*]]) – A list of RefinementFilter classes that implement filters on possible statement refinements. For details on how to construct such a filter, see the documentation of *indra.preassembler.refinement.RefinementFilter*. If no user-supplied filters are provided, the default ontology-based filter is applied. If a list of filters is provided here, the *indra.preassembler.refinement.OntologyRefinementFilter* isn't appended by default, and should be added by the user, if necessary. Default: None

Returns The returned list contains Statements representing the more concrete/refined versions of the Statements involving particular entities. The attribute *related_stmts* is also set to this list. However, if return_toplevel is False then all statements are returned, irrespective of level of specificity. In this case the relationships between statements can be accessed via the supports/supported_by attributes.

Return type list of `indra.statement.Statement`

Examples

A more general statement with no information about a Phosphorylation site is identified as supporting a more specific statement:

```
>>> from indra.ontology.bio import bio_ontology
>>> braf = Agent('BRAF')
>>> map2k1 = Agent('MAP2K1')
>>> st1 = Phosphorylation(braf, map2k1)
>>> st2 = Phosphorylation(braf, map2k1, residue='S')
>>> pa = Preassembler(bio_ontology, [st1, st2])
>>> combined_stmts = pa.combine_related()
>>> combined_stmts
[Phosphorylation(BRAF(), MAP2K1(), S)]
>>> combined_stmts[0].supported_by
[Phosphorylation(BRAF(), MAP2K1())]
>>> combined_stmts[0].supported_by[0].supports
[Phosphorylation(BRAF(), MAP2K1(), S)]
```

find_contradicts()

Return pairs of contradicting Statements.

Returns **contradicts** – A list of Statement pairs that are contradicting.

Return type list(tuple(*Statement*, *Statement*))

normalize_equivalences(*ns*, *rank_key=None*)

Normalize to one of a set of equivalent concepts across statements.

This function changes Statements in place without returning a value.

Parameters

- **ns** (*str*) – The db_refs namespace for which the equivalence relation should be applied.
- **rank_key** (*Optional[function]*) – A function handle which assigns a sort key to each entry in the given namespace to allow prioritizing in a controlled way which concept is normalized to.

normalize_opposites(*ns*, *rank_key=None*)

Normalize to one of a pair of opposite concepts across statements.

This function changes Statements in place without returning a value.

Parameters

- **ns** (*str*) – The db_refs namespace for which the opposite relation should be applied.
- **rank_key** (*Optional[function]*) – A function handle which assigns a sort key to each entry in the given namespace to allow prioritizing in a controlled way which concept is normalized to.

indra.preassembler.**find_refinements_for_statement**(*stmt*, *filters*)

Return refinements for a single statement given initialized filters.

Parameters

- **stmt** (*indra.statements.Statement*) – The statement whose relations should be found.
- **filters** (list[*indra.preassembler.refinement.RefinementFilter*]) – A list of refinement filter instances. The filters passed to this function need to have been initialized with stmts_by_hash.

Returns A set of statement hashes that this statement refines.

Return type set

indra.preassembler.**flatten_evidence**(*stmts*, *collect_from=None*)

Add evidence from *supporting* stmts to evidence for *supported* stmts.

Parameters

- **stmts** (list of indra.statements.Statement) – A list of top-level statements with associated supporting statements resulting from building a statement hierarchy with combine_related().
- **collect_from** (*str in ('supports', 'supported_by')*) – String indicating whether to collect and flatten evidence from the *supports* attribute of each statement or the *supported_by* attribute. If not set, defaults to 'supported_by'.

Returns **stmts** – Statement hierarchy identical to the one passed, but with the evidence lists for each statement now containing all of the evidence associated with the statements they are supported by.

Return type list of indra.statements.Statement

Examples

Flattening evidence adds the two pieces of evidence from the supporting statement to the evidence list of the top-level statement:

```
>>> from indra.ontology.bio import bio_ontology
>>> braf = Agent('BRAF')
>>> map2k1 = Agent('MAP2K1')
>>> st1 = Phosphorylation(braf, map2k1,
... evidence=[Evidence(text='foo'), Evidence(text='bar')])
>>> st2 = Phosphorylation(braf, map2k1, residue='S',
... evidence=[Evidence(text='baz'), Evidence(text='bak')])
>>> pa = Preassembler(bio_ontology, [st1, st2])
>>> pa.combine_related()
[Phosphorylation(BRAF(), MAP2K1(), S)]
>>> [e.text for e in pa.related_stmts[0].evidence]
['baz', 'bak']
>>> flattened = flatten_evidence(pa.related_stmts)
>>> sorted([e.text for e in flattened[0].evidence])
['bak', 'bar', 'baz', 'foo']
```

indra.preassembler.**flatten_stmts**(*stmts*)

Return the full set of unique stms in a pre-assembled stmt graph.

The flattened list of statements returned by this function can be compared to the original set of unique statements to make sure no statements have been lost during the preassembly process.

Parameters **stmts** (list of indra.statements.Statement) – A list of top-level statements with associated supporting statements resulting from building a statement hierarchy with combine_related().

Returns **stmts** – List of all statements contained in the hierarchical statement graph.

Return type list of indra.statements.Statement

Examples

Calling `combine_related()` on two statements results in one top-level statement; calling *flatten_stmts()* recovers both:

```
>>> from indra.ontology.bio import bio_ontology
>>> braf = Agent('BRAF')
>>> map2k1 = Agent('MAP2K1')
>>> st1 = Phosphorylation(braf, map2k1)
>>> st2 = Phosphorylation(braf, map2k1, residue='S')
>>> pa = Preassembler(bio_ontology, [st1, st2])
>>> pa.combine_related()
[Phosphorylation(BRAF(), MAP2K1(), S)]
>>> flattened = flatten_stmts(pa.related_stmts)
>>> flattened.sort(key=lambda x: x.matches_key())
>>> flattened
[Phosphorylation(BRAF(), MAP2K1()), Phosphorylation(BRAF(), MAP2K1(), S)]
```

indra.preassembler.**render_stmt_graph**(*statements*, *reduce=True*, *english=False*, *rankdir=None*, *agent_style=None*)

Render the statement hierarchy as a pygraphviz graph.

Parameters

- **statements** (list of `indra.statements.Statement`) – A list of top-level statements with associated supporting statements resulting from building a statement hierarchy with `combine_related()`.
- **reduce** (*bool*) – Whether to perform a transitive reduction of the edges in the graph. Default is True.
- **english** (*bool*) – If True, the statements in the graph are represented by their English-assembled equivalent; otherwise they are represented as text-formatted Statements.
- **rankdir** (*str or None*) – Argument to pass through to the pygraphviz *AGraph* constructor specifying graph layout direction. In particular, a value of 'LR' specifies a left-to-right direction. If None, the pygraphviz default is used.
- **agent_style** (*dict or None*) – Dict of attributes specifying the visual properties of nodes. If None, the following default attributes are used:

```
agent_style = {'color': 'lightgray', 'style': 'filled',
               'fontname': 'arial'}
```

Returns Pygraphviz graph with nodes representing statements and edges pointing from supported statements to supported_by statements.

Return type pygraphviz.AGraph

Examples

Pattern for getting statements and rendering as a Graphviz graph:

```
>>> from indra.ontology.bio import bio_ontology
>>> braf = Agent('BRAF')
>>> map2k1 = Agent('MAP2K1')
>>> st1 = Phosphorylation(braf, map2k1)
>>> st2 = Phosphorylation(braf, map2k1, residue='S')
>>> pa = Preassembler(bio_ontology, [st1, st2])
>>> pa.combine_related()
[Phosphorylation(BRAF(), MAP2K1(), S)]
>>> graph = render_stmt_graph(pa.related_stmts)
>>> graph.write('example_graph.dot') # To make the DOT file
>>> graph.draw('example_graph.png', prog='dot') # To make an image
```

Resulting graph:

4.6.2 Refinement filter classes and functions (indra.preassembler.refinement)

This module implements classes and functions that are used for finding refinements between INDRA Statements as part of the knowledge-assembly process. These are imported by the preassembler module.

class indra.preassembler.refinement.**OntologyRefinementFilter**(*ontology*)

This filter uses an ontology to position statements and their agents to filter down significantly on the set of possible relations for a given statement.

Parameters **ontology** (*indra.ontology.OntologyGraph*) – An INDRA ontology graph.

extend(*stmts_by_hash*)

Extend the initial data structures with a set of new statements.

Parameters **stmts_by_hash** (*dict[int, indra.statements.Statement]*) – A dict of statements keyed by their hashes.

get_related(*stmt*, *possibly_related=None*, *direction='less_specific'*)

Return a set of statement hashes that a given statement is potentially related to.

Parameters

- **stmt** (*indra.statements.Statement*) – The INDRA statement whose potential relations we want to filter.
- **possibly_related** (*set or None*) – A set of statement hashes that this statement is potentially related to, as determined by some other filter. If this parameter is a set (including an empty set), this function should return a subset of it (intuitively, this filter can only further eliminate some of the potentially related hashes that were previously determined to be potential relations). If this argument is None, the function must assume that no previous filter was run before, and should therefore return all the possible relations that it determines.
- **direction** (*str*) – One of 'less_specific' or 'more_specific. Since refinements are directed relations, this function can operate in two different directions: it can either find less specific potentially related stateemnts, or it can find more specific potentially related statements, as determined by this argument.

Returns A set of INDRA Statement hashes that are potentially related to the given statement.

Return type set of int

initialize(*stmts_by_hash*)
Initialize the filter class with a set of statements.

The filter can build up some useful data structures in this function before being applied to any specific statements.

Parameters **stmts_by_hash** (*dict[int, indra.statements.Statement]*) – A dict of statements keyed by their hashes.

class indra.preassembler.refinement.**RefinementConfirmationFilter**(*ontology*, *refinement_fun=None*)
This class runs the refinement function between potentially related statements to confirm whether they are indeed, conclusively in a refinement relationship with each other.

In this sense, this isn't a real filter, though implementing it as one is convenient. This filter is meant to be used as the final component in a series of pre-filters.

get_related(*stmt*, *possibly_related=None*, *direction='less_specific'*)
Return a set of statement hashes that a given statement is potentially related to.

Parameters

- **stmt** (*indra.statements.Statement*) – The INDRA statement whose potential relations we want to filter.
- **possibly_related** (*set or None*) – A set of statement hashes that this statement is potentially related to, as determined by some other filter. If this parameter is a set (including an empty set), this function should return a subset of it (intuitively, this filter can only further eliminate some of the potentially related hashes that were previously determined to be potential relations). If this argument is None, the function must assume that no previous filter was run before, and should therefore return all the possible relations that it determines.
- **direction** (*str*) – One of 'less_specific' or 'more_specific. Since refinements are directed relations, this function can operate in two different directions: it can either find less specific potentially related stateemnts, or it can find more specific potentially related statements, as determined by this argument.

Returns A set of INDRA Statement hashes that are potentially related to the given statement.

Return type set of int

class indra.preassembler.refinement.**RefinementFilter**
A filter which is applied to one or more statements to eliminate candidate refinements that are not possible according to some criteria. By applying a series of such filters, the preassembler can avoid doing n-by-n comparisons to determine refinements among n statements.

The filter class can take any number of constructor arguments that it needs to perform its task. The base class' constructor initializes a shared_data attribute as an empty dict.

It also needs to implement an initialize function which is called with a stmts_by_hash argument, containing a dict of statements keyed by hash. This function can build any data structures that may be needed to efficiently apply the filter later. It cab store any such data structures in the shared_data dict to be accessed by other functions later.

Finally, the class needs to implement a get_related function, which takes a single INDRA Statement as input to return the hashes of potentially related other statements that the filter was initialized with. The function also needs to take a possibly_related argument which is either None (no other filter was run before) or a set, which is the superset of possible relations as determined by some other previously applied filter.

extend(*stmts_by_hash*)
Extend the initial data structures with a set of new statements.

Parameters **stmts_by_hash** (*dict[int, indra.statements.Statement]*) – A dict of statements keyed by their hashes.

get_less_specifics(*stmt*, *possibly_related=None*)
Return a set of hashes of statements that are potentially related and less specific than the given statement.

get_more_specifics(*stmt*, *possibly_related=None*)
Return a set of hashes of statements that are potentially related and more specific than the given statement.

get_related(*stmt*, *possibly_related=None*, *direction='less_specific'*)
Return a set of statement hashes that a given statement is potentially related to.

Parameters

- **stmt** (*indra.statements.Statement*) – The INDRA statement whose potential relations we want to filter.
- **possibly_related** (*set or None*) – A set of statement hashes that this statement is potentially related to, as determined by some other filter. If this parameter is a set (including an empty set), this function should return a subset of it (intuitively, this filter can only further eliminate some of the potentially related hashes that were previously determined to be potential relations). If this argument is None, the function must assume that no previous filter was run before, and should therefore return all the possible relations that it determines.
- **direction** (*str*) – One of 'less_specific' or 'more_specific. Since refinements are directed relations, this function can operate in two different directions: it can either find less specific potentially related stateemnts, or it can find more specific potentially related statements, as determined by this argument.

Returns A set of INDRA Statement hashes that are potentially related to the given statement.

Return type set of int

initialize(*stmts_by_hash*)
Initialize the filter class with a set of statements.

The filter can build up some useful data structures in this function before being applied to any specific statements.

Parameters **stmts_by_hash** (*dict[int, indra.statements.Statement]*) – A dict of statements keyed by their hashes.

class indra.preassembler.refinement.**SplitGroupFilter**(*split_groups*)
This filter implements splitting statements into two groups and only considering refinement relationships between the groups but not within them.

get_related(*stmt*, *possibly_related=None*, *direction='less_specific'*)
Return a set of statement hashes that a given statement is potentially related to.

Parameters

- **stmt** (*indra.statements.Statement*) – The INDRA statement whose potential relations we want to filter.
- **possibly_related** (*set or None*) – A set of statement hashes that this statement is potentially related to, as determined by some other filter. If this parameter is a set (including an empty set), this function should return a subset of it (intuitively, this filter can only further eliminate some of the potentially related hashes that were previously determined to be potential relations). If this argument is None, the function must assume that no previous filter was run before, and should therefore return all the possible relations that it determines.

- **direction** (*str*) – One of 'less_specific' or 'more_specific. Since refinements are directed relations, this function can operate in two different directions: it can either find less specific potentially related stateemnts, or it can find more specific potentially related statements, as determined by this argument.

Returns A set of INDRA Statement hashes that are potentially related to the given statement.

Return type set of int

indra.preassembler.refinement.**get_agent_key**(*agent*)
Return a key for an Agent for use in refinement finding.

Parameters **agent** (*indra.statements.Agent or None*) – An INDRA Agent whose key should be returned.

Returns The key that maps the given agent to the ontology, with special handling for ungrounded and None Agents.

Return type tuple or None

indra.preassembler.refinement.**get_relevant_keys**(*agent_key*, *all_keys_for_role*, *ontology*, *direction*)
Return relevant agent keys for an agent key for refinement finding.

Parameters

- **agent_key** (*tuple or None*) – An agent key of interest.
- **all_keys_for_role** (*set*) – The set of all agent keys in a given statement corpus with a role matching that of the given agent_key.
- **ontology** (*indra.ontology.IndraOntology*) – An IndraOntology instance with respect to which relevant other agent keys are found for the purposes of refinement.
- **direction** (*str*) – The direction in which to find relevant agents. The two options are 'less_specific' and 'more_specific' for agents that are less and more specific, per the ontology, respectively.

Returns The set of relevant agent keys which this given agent key can possibly refine.

Return type set

4.6.3 Custom preassembly functions (indra.preassembler.custom_preassembly)

This module contains a library of functions that are useful for building custom preassembly logic for some applications. They are typically used as matches_fun or refinement_fun arguments to the Preassembler and other modules.

indra.preassembler.custom_preassembly.**agent_grounding_matches**(*agent*)
Return an Agent matches key just based on grounding, not state.

indra.preassembler.custom_preassembly.**agent_name_matches**(*agent*)
Return a sorted, normalized bag of words as the name.

indra.preassembler.custom_preassembly.**agent_name_polarity_matches**(*stmt*, *sign_dict*)
Return a key for normalized agent names and polarity.

indra.preassembler.custom_preassembly.**agent_name_stmt_matches**(*stmt*)
Return the normalized agent names.

indra.preassembler.custom_preassembly.**agent_name_stmt_type_matches**(*stmt*)
Return True if the statement type and normalized agent name matches.

indra.preassembler.custom_preassembly.**agents_stmt_type_matches**(*stmt*)
Return a matches key just based on Agent grounding and Stmt type.

4.6.4 Entity grounding mapping and standardization (indra.preassembler.grounding_mapper)

Grounding mapping

class indra.preassembler.grounding_mapper.mapper.**GroundingMapper**(*grounding_map=None*, *agent_map=None*, *ignores=None*, *misgrounding_map=None*, *use_adeft=True*, *gilda_mode=None*)

Maps grounding of INDRA Agents based on a given grounding map.

Each parameter, if not provided will result in loading the corresponding built-in grounding resource. To explicitly avoid loading the default, pass in an empty data structure as the given parameter, e.g., ignores=[].

Parameters

- **grounding_map** (*Optional[dict]*) – The grounding map, a dictionary mapping strings (entity names) to a dictionary of database identifiers.
- **agent_map** (*Optional[dict]*) – A dictionary mapping strings to grounded INDRA Agents with given state.
- **ignores** (*Optional[list]*) – A list of entity strings that, if encountered will result in the corresponding Statement being discarded.
- **misgrounding_map** (*Optional[dict]*) – A mapping dict similar to the grounding map which maps entity strings to a given grounding which is known to be incorrect and should be removed if encountered (making the remaining Agent ungrounded).
- **use_adeft** (*Optional[bool]*) – If True, Adeft will be attempted to be used for disambiguation of acronyms. Default: True
- **gilda_mode** (*Optional[str]*) – If None, Gilda will not be used at all. If 'web', the GILDA_URL setting from the config file or as an environmental variable is assumed to be the web service endpoint through which Gilda is used. If 'local', we assume that the gilda Python package is installed and will be used.

static check_grounding_map(*gm*)

Run sanity checks on the grounding map, raise error if needed.

map_agent(*agent*, *do_rename*)

Return the given Agent with its grounding mapped.

This function grounds a single agent. It returns the new Agent object (which might be a different object if we load a new agent state from json) or the same object otherwise.

Parameters

- **agent** (indra.statements.Agent) – The Agent to map.
- **do_rename** (*bool*) – If True, the Agent name is updated based on the mapped grounding. If do_rename is True the priority for setting the name is FamPlex ID, HGNC symbol, then the gene name from Uniprot.

Returns **grounded_agent** – The grounded Agent.

Return type indra.statements.Agent

map_agents_for_stmt(*stmt*, *do_rename=True*)

Return a new Statement whose agents have been grounding mapped.

Parameters

- **stmt** (indra.statements.Statement) – The Statement whose agents need mapping.
- **do_rename** (*Optional[bool]*) – If True, the Agent name is updated based on the mapped grounding. If do_rename is True the priority for setting the name is FamPlex ID, HGNC symbol, then the gene name from Uniprot. Default: True

Returns **mapped_stmt** – The mapped Statement.

Return type indra.statements.Statement

map_stmts(*stmts*, *do_rename=True*)

Return a new list of statements whose agents have been mapped

Parameters

- **stmts** (list of indra.statements.Statement) – The statements whose agents need mapping
- **do_rename** (*Optional[bool]*) – If True, the Agent name is updated based on the mapped grounding. If do_rename is True the priority for setting the name is FamPlex ID, HGNC symbol, then the gene name from Uniprot. Default: True

Returns **mapped_stmts** – A list of statements given by mapping the agents from each statement in the input list

Return type list of indra.statements.Statement

static rename_agents(*stmts*)

Return a list of mapped statements with updated agent names.

Creates a new list of statements without modifying the original list.

Parameters **stmts** (list of indra.statements.Statement) – List of statements whose Agents need their names updated.

Returns **mapped_stmts** – A new list of Statements with updated Agent names

Return type list of indra.statements.Statement

static standardize_agent_name(*agent*, *standardize_refs=True*)

Standardize the name of an Agent based on grounding information.

If an agent contains a FamPlex grounding, the FamPlex ID is used as a name. Otherwise if it contains a Uniprot ID, an attempt is made to find the associated HGNC gene name. If one can be found it is used as the agent name and the associated HGNC ID is added as an entry to the db_refs. Similarly, CHEBI, MESH and GO IDs are used in this order of priority to assign a standardized name to the Agent. If no relevant IDs are found, the name is not changed.

Parameters

- **agent** (*indra.statements.Agent*) – An INDRA Agent whose name attribute should be standardized based on grounding information.
- **standardize_refs** (*Optional[bool]*) – If True, this function assumes that the Agent's db_refs need to be standardized, e.g., HGNC mapped to UP. Default: True

static standardize_db_refs(*db_refs*)

Return a standardized db refs dict for a given db refs dict.

Parameters **db_refs** (*dict*) – A dict of db refs that may not be standardized, i.e., may be missing an available UP ID corresponding to an existing HGNC ID.

Returns The db_refs dict with standardized entries.

Return type dict

update_agent_db_refs(*agent*, *db_refs*, *do_rename=True*)
Update db_refs of agent using the grounding map

If the grounding map is missing one of the HGNC symbol or Uniprot ID, attempts to reconstruct one from the other.

Parameters

- **agent** (indra.statements.Agent) – The agent whose db_refs will be updated
- **db_refs** (*dict*) – The db_refs so set for the agent.
- **do_rename** (*Optional[bool]*) – If True, the Agent name is updated based on the mapped grounding. If do_rename is True the priority for setting the name is FamPlex ID, HGNC symbol, then the gene name from Uniprot. Default: True

indra.preassembler.grounding_mapper.mapper.**load_grounding_map**(*grounding_map_path*, *lineterminator='\r\n'*, *hgnc_symbols=True*)
Return a grounding map dictionary loaded from a csv file.

In the file pointed to by grounding_map_path, the number of name_space ID pairs can vary per row and commas are used to pad out entries containing fewer than the maximum amount of name spaces appearing in the file. Lines should be terminated with

both a carriage return and a new line by default.

Optionally, one can specify another csv file (pointed to by ignore_path) containing agent texts that are degenerate and should be filtered out.

It is important to note that this function assumes that the mapping file entries for the HGNC key are symbols not IDs. These symbols are converted to IDs upon loading here.

Parameters

- **grounding_map_path** (*str*) – Path to csv file containing grounding map information. Rows of the file should be of the form <agent_text>,<name_space_1>,<ID_1>,... <name_space_n>,<ID_n>
- **lineterminator** (*Optional[str]*) – Line terminator used in input csv file. Default:
- **hgnc_symbols** (*Optional[bool]*) – Set to True if the grounding map file contains HGNC symbols rather than IDs. In this case, the entries are replaced by IDs. Default: True

Returns **g_map** – The grounding map constructed from the given files.

Return type dict

Disambiguation with machine-learned models

class indra.preassembler.grounding_mapper.disambiguate.**DisambManager**
Manages running of disambiguation models

Has methods to run disambiguation with either adeft or gilda. Each instance of this class uses a single database connection.

run_adeft_disambiguation(*stmt*, *agent*, *idx*, *agent_txt*)
Run Adeft disambiguation on an Agent in a given Statement.

This function looks at the evidence of the given Statement and attempts to look up the full paper or the abstract for the evidence. If both of those fail, the evidence sentence itself is used for disambiguation.

The disambiguation model corresponding to the Agent text is then called, and the highest scoring returned grounding is set as the Agent's new grounding.

The Statement's annotations as well as the Agent are modified in place and no value is returned.

Parameters

- **stmt** (*indra.statements.Statement*) – An INDRA Statement in which the Agent to be disambiguated appears.
- **agent** (*indra.statements.Agent*) – The Agent (potentially grounding mapped) which we want to disambiguate in the context of the evidence of the given Statement.
- **idx** (*int*) – The index of the new Agent's position in the Statement's agent list (needed to set annotations correctly).

Returns True if disambiguation was successfully applied, and False otherwise. Reasons for a False response can be the lack of evidence as well as failure to obtain text for grounding disambiguation.

Return type bool

run_gilda_disambiguation(*stmt*, *agent*, *idx*, *agent_txt*, *mode='web'*)

Run Gilda disambiguation on an Agent in a given Statement.

This function looks at the evidence of the given Statement and attempts to look up the full paper or the abstract for the evidence. If both of those fail, the evidence sentence itself is used for disambiguation. The disambiguation model corresponding to the Agent text is then called, and the highest scoring returned grounding is set as the Agent's new grounding.

The Statement's annotations as well as the Agent are modified in place and no value is returned.

Parameters

- **stmt** (*indra.statements.Statement*) – An INDRA Statement in which the Agent to be disambiguated appears.
- **agent** (*indra.statements.Agent*) – The Agent (potentially grounding mapped) which we want to disambiguate in the context of the evidence of the given Statement.
- **idx** (*int*) – The index of the new Agent's position in the Statement's agent list (needed to set annotations correctly).
- **mode** (*Optional[str]*) – If 'web', the web service given in the GILDA_URL config setting or environmental variable is used. Otherwise, the gilda package is attempted to be imported and used. Default: web

Returns True if disambiguation was successfully applied, and False otherwise. Reasons for a False response can be the lack of evidence as well as failure to obtain text for grounding disambiguation.

Return type bool

Gilda grounding functions

This module implements a client to the Gilda grounding web service, and contains functions to help apply it during the course of INDRA assembly.

indra.preassembler.grounding_mapper.gilda.**get_gilda_models**(*mode='web'*)
Return a list of strings for which Gilda has a disambiguation model.

Parameters **mode** (*Optional[str]*) – If 'web', the web service given in the GILDA_URL config setting or environmental variable is used. Otherwise, the gilda package is attempted to be imported and used. Default: web

Returns A list of entity strings.

Return type list[str]

indra.preassembler.grounding_mapper.gilda.**get_grounding**(*txt*, *context=None*, *mode='web'*)
Return the top Gilda grounding for a given text.

Parameters

- **txt** (*str*) – The text to ground.
- **context** (*Optional[str]*) – Any context for the grounding.
- **mode** (*Optional[str]*) – If 'web', the web service given in the GILDA_URL config setting or environmental variable is used. Otherwise, the gilda package is attempted to be imported and used. Default: web

Return type Tuple[Mapping[str, Any], List[Any]]

Returns

- *dict* – If no grounding was found, it is an empty dict. Otherwise, it's a dict with the top grounding returned from Gilda.
- *list* – The list of ScoredMatches

indra.preassembler.grounding_mapper.gilda.**ground_agent**(*agent*, *txt*, *context=None*, *mode='web'*)
Set the grounding of a given agent, by re-grounding with Gilda.

This function changes the agent in place without returning a value.

Parameters

- **agent** (*indra.statements.Agent*) – The Agent whose db_refs shuld be changed.
- **txt** (*str*) – The text by which the Agent should be grounded.
- **context** (*Optional[str]*) – Any additional text context to help disambiguate the sense associated with txt.
- **mode** (*Optional[str]*) – If 'web', the web service given in the GILDA_URL config setting or environmental variable is used. Otherwise, the gilda package is attempted to be imported and used. Default: web

indra.preassembler.grounding_mapper.gilda.**ground_statement**(*stmt*, *mode='web'*, *ungrounded_only=False*)
Set grounding for Agents in a given Statement using Gilda.

This function modifies the original Statement/Agents in place.

Parameters

- **stmt** (*indra.statements.Statement*) – A Statement to ground

- **mode** (*Optional[str]*) – If 'web', the web service given in the GILDA_URL config setting or environmental variable is used. Otherwise, the gilda package is attempted to be imported and used. Default: web
- **ungrounded_only** (*Optional[str]*) – If True, only ungrounded Agents will be grounded, and ones that are already grounded will not be modified. Default: False

indra.preassembler.grounding_mapper.gilda.**ground_statements**(*stmts*, *mode='web'*, *sources=None*, *ungrounded_only=False*)

Set grounding for Agents in a list of Statements using Gilda.

This function modifies the original Statements/Agents in place.

Parameters

- **stmts** (*list[indra.statements.Statement]*) – A list of Statements to ground
- **mode** (*Optional[str]*) – If 'web', the web service given in the GILDA_URL config setting or environmental variable is used. Otherwise, the gilda package is attempted to be imported and used. Default: web
- **sources** (*Optional[list]*) – If given, only statements from the given sources are grounded. The sources have to correspond to valid source_api entries, e.g., 'reach', 'sparser', etc. If not given, statements from all sources are grounded.
- **ungrounded_only** (*Optional[str]*) – If True, only ungrounded Agents will be grounded, and ones that are already grounded will not be modified. Default: False

Returns The list of Statements that were changed in place by reference.

Return type list[indra.statement.Statements]

Analysis scripts for grounding

indra.preassembler.grounding_mapper.analysis.**agent_texts**(*agents*)

Return a list of all agent texts from a list of agents.

None values are associated to agents without agent texts

Parameters **agents** (list of `indra.statements.Agent`) –

Returns agent texts from input list of agents

Return type list of str/None

indra.preassembler.grounding_mapper.analysis.**agent_texts_with_grounding**(*stmts*)

Return agent text groundings in a list of statements with their counts

Parameters **stmts** (list of `indra.statements.Statement`) –

Returns

List of tuples of the form (text: str, ((name_space: str, ID: str, count: int)...), total_count: int)

Where the counts within the tuple of groundings give the number of times an agent with the given agent_text appears grounded with the particular name space and ID. The total_count gives the total number of times an agent with text appears in the list of statements.

Return type list of tuple

indra.preassembler.grounding_mapper.analysis.**all_agents**(*stmts*)

Return a list of all of the agents from a list of statements.

Only agents that are not None and have a TEXT entry are returned.

Parameters **stmts** (list of indra.statements.Statement) –

Returns **agents** – List of agents that appear in the input list of indra statements.

Return type list of indra.statements.Agent

indra.preassembler.grounding_mapper.analysis.**get_agents_with_name**(*name*, *stmts*)
Return all agents within a list of statements with a particular name.

indra.preassembler.grounding_mapper.analysis.**get_sentences_for_agent**(*text*, *stmts*, *max_sentences=None*)
Returns evidence sentences with a given agent text from a list of statements.

Parameters

- **text** (*str*) – An agent text
- **stmts** (list of indra.statements.Statement) – INDRA Statements to search in for evidence statements.
- **max_sentences** (*Optional[int/None]*) – Cap on the number of evidence sentences to return. Default: None

Returns **sentences** – Evidence sentences from the list of statements containing the given agent text.

Return type list of str

indra.preassembler.grounding_mapper.analysis.**protein_map_from_twg**(*twg*)
Build map of entity texts to validate protein grounding.

Looks at the grounding of the entity texts extracted from the statements and finds proteins where there is grounding to a human protein that maps to an HGNC name that is an exact match to the entity text. Returns a dict that can be used to update/expand the grounding map.

Parameters **twg** (*list of tuple*) – list of tuples of the form output by agent_texts_with_grounding

Returns **protein_map** – dict keyed on agent text with associated values {'TEXT': agent_text, 'UP': uniprot_id}. Entries are for agent texts where the grounding map was able to find human protein grounded to this agent_text in Uniprot.

Return type dict

indra.preassembler.grounding_mapper.analysis.**save_base_map**(*filename*, *grouped_by_text*)
Dump a list of agents along with groundings and counts into a csv file

Parameters

- **filename** (*str*) – Filepath for output file
- **grouped_by_text** (*list of tuple*) – List of tuples of the form output by agent_texts_with_grounding

indra.preassembler.grounding_mapper.analysis.**save_sentences**(*twg*, *stmts*, *filename*, *agent_limit=300*)
Write evidence sentences for stmts with ungrounded agents to csv file.

Parameters

- **twg** (*list of tuple*) – list of tuples of ungrounded agent_texts with counts of the number of times they are mentioned in the list of statements. Should be sorted in descending order by the counts. This is of the form output by the function ungrounded texts.
- **stmts** (list of indra.statements.Statement) –
- **filename** (*str*) – Path to output file

- **agent_limit** (*Optional[int]*) – Number of agents to include in output file. Takes the top agents by count.

indra.preassembler.grounding_mapper.analysis.**ungrounded_texts**(*stmts*)
Return a list of all ungrounded entities ordered by number of mentions

Parameters **stmts** (list of indra.statements.Statement) –

Returns **ungroundc** – list of tuples of the form (text: str, count: int) sorted in descending order by count.

Return type list of tuple

4.6.5 Site curation and mapping (indra.preassembler.sitemapper)

class indra.preassembler.sitemapper.**MappedStatement**(*original_stmt*, *mapped_mods*, *mapped_stmt*)
Information about a Statement found to have invalid sites.

Parameters

- **original_stmt** (indra.statements.Statement) – The statement prior to mapping.
- **mapped_mods** (*list of MappedSite*) – A list of MappedSite objects.
- **mapped_stmt** (indra.statements.Statement) – The statement after mapping. Note that if no information was found in the site map, it will be identical to the original statement.

class indra.preassembler.sitemapper.**SiteMapper**(*site_map=None*, *use_cache=False*, *cache_path=None*, *do_methionine_offset=True*, *do_orthology_mapping=True*, *do_isoform_mapping=True*)
Use site information to fix modification sites in Statements.

This is a wrapper around the protmapper package's ProtMapper class and adds all the additional functionality to handle INDRA Statements and Agents.

Parameters

- **site_map** (dict (as returned by load_site_map())) – A dict mapping tuples of the form *(gene, orig_res, orig_pos)* to a tuple of the form *(correct_res, correct_pos, comment)*, where *gene* is the string name of the gene (canonicalized to HGNC); *orig_res* and *orig_pos* are the residue and position to be mapped; *correct_res* and *correct_pos* are the corrected residue and position, and *comment* is a string describing the reason for the mapping (species error, isoform error, wrong residue name, etc.).
- **use_cache** (*Optional[bool]*) – If True, the SITEMAPPER_CACHE_PATH from the config (or environment) is loaded and cached mappings are read and written to the given path. Otherwise, no cache is used. Default: False
- **do_methionine_offset** (*boolean*) – Whether to check for off-by-one errors in site position (possibly) attributable to site numbering from mature proteins after cleavage of the initial methionine. If True, checks the reference sequence for a known modification at 1 site position greater than the given one; if there exists such a site, creates the mapping. Default is True.
- **do_orthology_mapping** (*boolean*) – Whether to check sequence positions for known modification sites in mouse or rat sequences (based on PhosphoSitePlus data). If a mouse/rat site is found that is linked to a site in the human reference sequence, a mapping is created. Default is True.

- **do_isoform_mapping** (*boolean*) – Whether to check sequence positions for known modifications in other human isoforms of the protein (based on PhosphoSitePlus data). If a site is found that is linked to a site in the human reference sequence, a mapping is created. Default is True.

Examples

Fixing site errors on both the modification state of an agent (MAP2K1) and the target of a Phosphorylation statement (MAPK1):

```
>>> map2k1_phos = Agent('MAP2K1', db_refs={'UP':'Q02750'}, mods=[
... ModCondition('phosphorylation', 'S', '217'),
... ModCondition('phosphorylation', 'S', '221')])
>>> mapk1 = Agent('MAPK1', db_refs={'UP':'P28482'})
>>> stmt = Phosphorylation(map2k1_phos, mapk1, 'T','183')
>>> (valid, mapped) = default_mapper.map_sites([stmt])
>>> valid
[]
>>> mapped
[
MappedStatement:
    original_stmt: Phosphorylation(MAP2K1(mods: (phosphorylation, S, 217),␣
↪(phosphorylation, S, 221)), MAPK1(), T, 183)
    mapped_mods: MappedSite(up_id='Q02750', error_code=None, valid=False, orig_res=
↪'S', orig_pos='217', mapped_id='Q02750', mapped_res='S', mapped_pos='218',␣
↪description='off by one', gene_name='MAP2K1')
                 MappedSite(up_id='Q02750', error_code=None, valid=False, orig_res=
↪'S', orig_pos='221', mapped_id='Q02750', mapped_res='S', mapped_pos='222',␣
↪description='off by one', gene_name='MAP2K1')
                 MappedSite(up_id='P28482', error_code=None, valid=False, orig_res=
↪'T', orig_pos='183', mapped_id='P28482', mapped_res='T', mapped_pos='185',␣
↪description='INFERRED_MOUSE_SITE', gene_name='MAPK1')
    mapped_stmt: Phosphorylation(MAP2K1(mods: (phosphorylation, S, 218),␣
↪(phosphorylation, S, 222)), MAPK1(), T, 185)
]
>>> ms = mapped[0]
>>> ms.original_stmt
Phosphorylation(MAP2K1(mods: (phosphorylation, S, 217), (phosphorylation, S, 221)),␣
↪MAPK1(), T, 183)
>>> ms.mapped_mods
[MappedSite(up_id='Q02750', error_code=None, valid=False, orig_res='S', orig_pos=
↪'217', mapped_id='Q02750', mapped_res='S', mapped_pos='218', description='off by␣
↪one', gene_name='MAP2K1'), MappedSite(up_id='Q02750', error_code=None,␣
↪valid=False, orig_res='S', orig_pos='221', mapped_id='Q02750', mapped_res='S',␣
↪mapped_pos='222', description='off by one', gene_name='MAP2K1'), MappedSite(up_id=
↪'P28482', error_code=None, valid=False, orig_res='T', orig_pos='183', mapped_id=
↪'P28482', mapped_res='T', mapped_pos='185', description='INFERRED_MOUSE_SITE',␣
↪gene_name='MAPK1')]
>>> ms.mapped_stmt
Phosphorylation(MAP2K1(mods: (phosphorylation, S, 218), (phosphorylation, S, 222)),␣
↪MAPK1(), T, 185)
```

map_sites(*stmts*)

Check a set of statements for invalid modification sites.

Statements are checked against Uniprot reference sequences to determine if residues referred to by post-translational modifications exist at the given positions.

If there is nothing amiss with a statement (modifications on any of the agents, modifications made in the statement, etc.), then the statement goes into the list of valid statements. If there is a problem with the statement, the offending modifications are looked up in the site map (`site_map`), and an instance of *MappedStatement* is added to the list of mapped statements.

Parameters **stmts** (list of `indra.statement.Statement`) – The statements to check for site errors.

Returns 2-tuple containing (valid_statements, mapped_statements). The first element of the tuple is a list of valid statements (`indra.statement.Statement`) that were not found to contain any site errors. The second element of the tuple is a list of mapped statements (*MappedStatement*) with information on the incorrect sites and corresponding statements with correctly mapped sites.

Return type tuple

4.7 Belief prediction (`indra.belief`)

4.7.1 Belief Engine API (`indra.belief`)

class indra.belief.**BayesianScorer**(*prior_counts*, *subtype_counts*)

This is a belief scorer which assumes a Beta prior and a set of prior counts of correct and incorrect instances for a given source. It exposes and interface to take additional counts and update its probability parameters which can then be used to calculate beliefs on a set of Statements.

Parameters

- **prior_counts** (`Dict[str, List[int]]`) – A dictionary of counts of the form [pos, neg] for each source.
- **subtype_counts** (`Dict[str, Dict[str, List[int]]]`) – A dict of dicts of counts of the form [pos, neg] for each subtype within a source.

update_counts(*prior_counts*, *subtype_counts*)

Update the internal counts based on given new counts.

Parameters

- **prior_counts** (`Dict[str, List[int]]`) – A dictionary of counts of the form [pos, neg] for each source.
- **subtype_counts** (`Dict[str, Dict[str, List[int]]]`) – A dict of dicts of counts of the form [pos, neg] for each subtype within a source.

Return type `None`

update_probs()

Update the internal probability values given the counts.

class indra.belief.**BeliefEngine**(*scorer=None*, *matches_fun=None*, *refinements_graph=None*)

Assigns beliefs to INDRA Statements based on supporting evidence.

Parameters

- **scorer** (`Optional[BeliefScorer]`) – A BeliefScorer object that computes the prior probability of a statement given its its statment type, evidence, or other features. Must implement the *score_statements* method which takes Statements and computes the belief score of a

statement, and the *check_prior_probs* method which takes a list of INDRA Statements and verifies that the scorer has all the information it needs to score every statement in the list, and raises an exception if not.

- **matches_fun** (Optional[Callable[[*Statement*], str]]) – A function handle for a custom matches key if a non-default one is used. Default is None.
- **refinements_graph** (Optional[DiGraph]) – A graph whose nodes are statement hashes, and edges point from a more specific to a less specific statement representing a refinement. If not given, a new graph is constructed here.

get_hierarchy_probs(*statements*)

Gets hierarchical belief probabilities for INDRA Statements.

Parameters **statements** (Sequence[*Statement*]) – A list of INDRA Statements whose belief scores are to be calculated. Each Statement object's belief attribute is updated by this function.

Return type Dict[int, float]

Returns A dictionary mapping statement hashes to corresponding belief scores. Hashes are calculated using the instance's *self.matches_fun*.

get_hierarchy_probs_from_hashes(*statements*, *refiners_list*)

Return the full belief of a statement with refiners given as hashes.

Parameters

- **statements** (Sequence[*Statement*]) – Statements to calculate beliefs for.
- **refiners_list** (List[List[int]]) – A list corresponding to the list of statements, where each entry is a list of statement hashes for the statements that are refinements (i.e., more specific versions) of the corresponding statement in the statements list. If there are no refiner statements the entry should be an empty list.

Return type Dict[int, float]

Returns A dictionary mapping statement hashes to corresponding belief scores.

set_hierarchy_probs(*statements*)

Sets hierarchical belief probabilities for INDRA Statements.

The Statements are assumed to be in a hierarchical relation graph with the supports and supported_by attribute of each Statement object having been set. The hierarchical belief probability of each Statement is calculated based the accumulated evidence from both itself and its more specific statements in the hierarchy graph.

Parameters **statements** (Sequence[*Statement*]) – A list of INDRA Statements whose belief scores are to be calculated. Each Statement object's belief attribute is updated by this function.

Return type None

set_linked_probs(*linked_statements*)

Sets the belief probabilities for a list of linked INDRA Statements.

The list of LinkedStatement objects is assumed to come from the MechanismLinker. The belief probability of the inferred Statement is assigned the joint probability of its source Statements.

Parameters **linked_statements** (List[*LinkedStatement*]) – A list of INDRA LinkedStatements whose belief scores are to be calculated. The belief attribute of the inferred Statement in the LinkedStatement object is updated by this function.

Return type None

set_prior_probs(*statements*)
Sets the prior belief probabilities for a list of INDRA Statements.

The Statements are assumed to be de-duplicated. In other words, each Statement in the list passed to this function is assumed to have a list of Evidence objects that support it. The prior probability of each Statement is calculated based on the number of Evidences it has and their sources.

Parameters **statements** (Sequence[*Statement*]) – A list of INDRA Statements whose belief scores are to be calculated. Each Statement object's belief attribute is updated by this function.

Return type None

class indra.belief.**BeliefScorer**
Base class for a belief engine scorer, which computes the prior probability of a statement given its type and evidence.

To use with the belief engine, make a subclass with methods implemented.

check_prior_probs(*statements*)
Make sure the scorer has all the information needed to compute belief scores of each statement in the provided list, and raises an exception otherwise.

Parameters **statements** (Sequence[*Statement*]) – List of statements to check

Return type None

score_statement(*statement*, *extra_evidence=None*)
Score a single statement by passing arguments to *score_statements*.

Return type float

score_statements(*statements*, *extra_evidence=None*)
Computes belief probabilities for a list of INDRA Statements.

The Statements are assumed to be de-duplicated. In other words, each Statement is assumed to have a list of Evidence objects that supports it. The probability of correctness of the Statement is generally calculated based on the number of Evidences it has, their sources, and other features depending on the subclass implementation.

Parameters

- **statements** (Sequence[*Statement*]) – INDRA Statements whose belief scores are to be calculated.
- **extra_evidence** (Optional[List[List[*Evidence*]]]) – A list corresponding to the given list of statements, where each entry is a list of Evidence objects providing additional support for the corresponding statement (i.e., Evidences that aren't already included in the Statement's own evidence list).

Return type Sequence[float]

Returns The computed prior probabilities for each statement.

class indra.belief.**SimpleScorer**(*prior_probs=None*, *subtype_probs=None*)
Computes the prior probability of a statement given its type and evidence.

Parameters

- **prior_probs** (Optional[Dict[str, Dict[str, float]]]) – A dictionary of prior probabilities used to override/extend the default ones. There are two types of prior probabilities: rand and syst, corresponding to random error and systematic error rate for each knowledge source. The prior_probs dictionary has the general structure {'rand': {'s1': pr1, ..., 'sn': prn}, 'syst': {'s1': ps1, ..., 'sn': psn}} where 's1' ... 'sn' are names of input sources and

pr1 ... prn and ps1 ... psn are error probabilities. Examples: {'rand': {'some_source': 0.1}} sets the random error rate for some_source to 0.1; {'rand': {''}}

- **subtype_probs** (Optional[Dict[str, Dict[str, float]]]) – A dictionary of random error probabilities for knowledge sources. When a subtype random error probability is not specified, will just use the overall type prior in prior_probs. If None, will only use the priors for each rule.

check_prior_probs(*statements*)

Throw Exception if BeliefEngine parameter is missing.

Make sure the scorer has all the information needed to compute belief scores of each statement in the provided list, and raises an exception otherwise.

Parameters **statements** (Sequence[*Statement*]) – List of statements to check.

Return type None

score_evidence_list(*evidences*)

Return belief score given a list of supporting evidences.

Parameters **evidences** (List[*Evidence*]) – List of evidences to use for calculating a statement's belief.

Return type float

Returns Belief value based on the evidences.

score_statements(*statements*, *extra_evidence=None*)

Computes belief probabilities for a list of INDRA Statements.

The Statements are assumed to be de-duplicated. In other words, each Statement is assumed to have a list of Evidence objects that supports it. The probability of correctness of the Statement is generally calculated based on the number of Evidences it has, their sources, and other features depending on the subclass implementation.

Parameters

- **statements** (Sequence[*Statement*]) – INDRA Statements whose belief scores are to be calculated.
- **extra_evidence** (Optional[List[List[*Evidence*]]]) – A list corresponding to the given list of statements, where each entry is a list of Evidence objects providing additional support for the corresponding statement (i.e., Evidences that aren't already included in the Statement's own evidence list).

Return type List[float]

Returns The computed prior probabilities for each statement.

update_probs(*prior_probs=None*, *subtype_probs=None*)

Update Scorer's prior probabilities with the given dictionaries.

Return type None

indra.belief.**assert_no_cycle**(*g*)

If the graph has cycles, throws AssertionError.

This can be used to make sure that a refinements graph is a DAG.

Parameters **g** (DiGraph) – A refinements graph.

Return type None

indra.belief.**build_refinements_graph**(*statements*, *matches_fun=None*)
Return a DiGraph based on matches hashes and Statement refinements.

Parameters

- **statements** (Sequence[*Statement*]) – A list of Statements with *supports* Statements, used to generate the refinements graph.
- **matches_fun** (Optional[Callable[[*Statement*], str]]) – An optional function to calculate the matches key and hash of a given statement. Default: None

Return type DiGraph

Returns A networkx graph whose nodes are statement hashes carrying a stmt attribute with the actual statement object. Edges point from less detailed to more detailed statements (i.e., from a statement to another statement that refines it).

indra.belief.**check_extra_evidence**(*extra_evidence*, *num_stmts*)
Check whether extra evidence list has correct length/contents.

Raises ValueError if the extra_evidence list does not match the length num_stmts, or if it contains items other than empty lists or lists of Evidence objects.

Parameters

- **extra_evidence** (Optional[List[List[*Evidence*]]]) – A list of length num_stmts where each entry is a list of Evidence objects, or None. If extra_evidence is None, the function returns without raising an error.
- **num_stmts** (int) – An integer giving the required length of the extra_evidence list (which should correspond to a list of statements)

Return type None

indra.belief.**evidence_random_noise_prior**(*evidence*, *type_probs*, *subtype_probs*)
Gets the random-noise prior probability for this evidence.

If the evidence corresponds to a subtype, and that subtype has a curated prior noise probability, use that.

Otherwise, gives the random-noise prior for the overall rule type.

Return type float

indra.belief.**extend_refinements_graph**(*g*, *stmt*, *less_specifics*, *matches_fun=None*)
Extend refinements graph with a new statement and its refinements.

Parameters

- **g** (DiGraph) – A refinements graph to be extended.
- **stmt** (*Statement*) – The statement to be added to the refinements graph.
- **less_specifics** (List[int]) – A list of statement hashes of statements that are refined by this statement (i.e., are less specific versions of it).
- **matches_fun** (Optional[Callable[[*Statement*], str]]) – An optional function to calculate the matches key and hash of a given statement. Default: None

Return type DiGraph

indra.belief.**get_ev_for_stmts_from_hashes**(*statements*, *refiners_list*, *refinements_graph*)
Collect evidence from the more specific statements of a list of statements.

Similar to *get_ev_for_stmts_from_supports()*, but the more specific statements are specified explicitly by the hashes in *refiners_list* rather than obtained from the *supports* attribute of each statement. In addition, the

refinements_graph argument is expected to have been pre-calculated (using the same matches key function used to generate the hashes in *refiners_list*) and hence is not optional.

Parameters

- **statements** (Sequence[*Statement*]) – A list of Statements with *supports* Statements.
- **refiners_list** (List[List[int]]) – A list corresponding to the list of statements, where each entry is a list of statement hashes for the statements that are refinements (i.e., more specific versions) of the corresponding statement in the statements list. If there are no refiner statements for a statement the entry should be an empty list.
- **refinements_graph** (DiGraph) – A networkx graph whose nodes are statement hashes carrying a stmt attribute with the actual statement object. Edges point from less detailed to more detailed statements (i.e., from a statement to another statement that refines it).

Return type List[List[*Evidence*]]

Returns A list corresponding to the given list of statements, where each entry is a list of Evidence objects providing additional support for the corresponding statement (i.e., Evidences that aren't already included in the Statement's own evidence list).

indra.belief.**get_ev_for_stmts_from_supports**(*statements*, *refinements_graph=None*, *matches_fun=None*)

Collect evidence from the more specific statements of a list of statements.

Parameters

- **statements** (Sequence[*Statement*]) – A list of Statements with *supports* Statements.
- **refinements_graph** (Optional[DiGraph]) – A networkx graph whose nodes are statement hashes carrying a stmt attribute with the actual statement object. Edges point from less detailed to more detailed statements (i.e., from a statement to another statement that refines it). If not provided, the graph is generated from *statements* using the function *build_refinements_graph()*.
- **matches_fun** (Optional[Callable[[*Statement*], str]]) – An optional function to calculate the matches key and hash of a given statement. If not provided, the default matches function is used. Default: None.

Return type List[List[*Evidence*]]

Returns A list corresponding to the given list of statements, where each entry is a list of Evidence objects providing additional support for the corresponding statement (i.e., Evidences that aren't already included in the Statement's own evidence list).

indra.belief.**get_ranked_stmts**(*g*)

Return a topological sort of statements from a graph.

indra.belief.**get_stmt_evidence**(*stmt*, *ix*, *extra_evidence*)

Combine a statements' own evidence with any extra evidence provided.

Return type List[*Evidence*]

indra.belief.**sample_statements**(*stmts*, *seed=None*)

Return statements sampled according to belief.

Statements are sampled independently according to their belief scores. For instance, a Statement with a belief score of 0.7 will end up in the returned Statement list with probability 0.7.

Parameters

- **stmts** (Sequence[*Statement*]) – A list of INDRA Statements to sample.
- **seed** (Optional[int]) – A seed for the random number generator used for sampling.

Return type List[*Statement*]

Returns A list of INDRA Statements that were chosen by random sampling according to their respective belief scores.

indra.belief.**tag_evidence_subtype**(*evidence*)

Returns the type and subtype of an evidence object as a string, typically the extraction rule or database from which the statement was generated.

For biopax, this is just the database name.

Parameters **statement** – The statement which we wish to subtype

Return type Tuple[str, Optional[str]]

Returns A tuple with (type, subtype), both strings. Returns (type, None) if the type of statement is not yet handled in this function.

4.7.2 Belief prediction with sklearn models (indra.belief.skl)

class indra.belief.skl.**CountsScorer**(*model*, *source_list*, *include_more_specific=False*, *use_stmt_type=False*, *use_num_members=False*, *use_num_pmids=False*, *use_promoter=False*, *use_avg_evidence_len=False*)

Belief model learned from evidence counts and other stmt properties.

If using a DataFrame for Statement data, it should have the following columns:

- *stmt_type*
- *source_counts*

Alternatively, if the DataFrame doesn't have a *source_counts* column, it should have columns with names matching the sources in *self.source_list*.

Parameters

- **model** (BaseEstimator) – Any instance of a classifier object supporting the methods *fit*, *predict_proba*, *predict*, and *predict_log_proba*.
- **source_list** (List[str]) – List of strings denoting the evidence sources (evidence.source_api values) to be used for prediction.
- **include_more_specific** (bool) – If True, will add extra columns to the statement data matrix for the source counts drawn from more specific evidences; if use_num_pmids is True, will also add an additional column for the number of PMIDs from more specific evidences. If False, these columns will not be included even if the *extra_evidence* argument is passed to the *stmts_to_matrix* method. This is to ensure that the featurization of statements is consistent between training and prediction.
- **use_stmt_type** (bool) – Whether to include statement type as a feature.
- **use_num_members** (bool) – Whether to include a feature denoting the number of members of the statement. Primarily for stratifying belief predictions about Complex statements with more than two members. Cannot be used for statement data passed in as a DataFrame.
- **use_num_pmids** (bool) – Whether to include a feature for the total number of unique PMIDs supporting each statement. Cannot be used for statement passed in as a DataFrame.
- **use_promoter** (bool) – Whether to include a feature giving the fraction of evidence (0 to 1) containing the (case-insensitive) word "promoter". Tends to improve misclassification of Complex statements that actually refer to protein-DNA binding.

- **use_avg_evidence_len** (bool) – Whether to include a feature giving the average evidence sentence length (in space-separated tokens).

Example

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression()
all_stmt_sources = CountsScorer.get_all_sources(stmts)
scorer = CountsScorer(clf, all_stmt_sources, use_stmt_type=True,
                      use_num_pmids=True)
scorer.fit(stmts, y_arr)
be = BeliefEngine(scorer)
be.set_hierarchy_probs(stmts)
```

df_to_matrix(*df*)

Convert a DataFrame of statement data to a feature matrix.

Based on information available in a DataFrame of statement data, this implementation uses only source counts and statement type in building a feature matrix, and will raise a ValueError if either *self.use_num_members* or *self.use_num_pmids* is set.

Features are encoded as follows:

- One column for every source listed in *self.source_list*, containing the number of statement evidences from that source. If *extra_evidence* is provided, these are used in combination with the Statement's own evidence in determining source counts.
- If *self.use_stmt_type* is set, statement type is included via one-hot encoding, with one column for each statement type.

Parameters **df** (DataFrame) – A pandas DataFrame with statement metadata. It should have columns *stmt_type* and *source_counts*; alternatively, if it doesn't have a *source_counts* column, it should have columns with names matching the sources in *self.source_list*.

Return type ndarray

Returns Feature matrix for the statement data.

static get_all_sources(*stmts*, *include_more_specific=True*, *include_less_specific=True*)

Get a list of all the source_apis supporting the given statements.

Useful for determining the set of sources to be used for fitting and prediction.

Parameters

- **stmts** (Sequence[*Statement*]) – A list of INDRA Statements to collect source APIs for.
- **include_more_specific** (bool) – If True (default), then includes the source APIs for the more specific statements in the *supports* attribute of each statement.
- **include_less_specific** (bool) – If True (default), then includes the source APIs for the less specific statements in the *supported_by* attribute of each statement.

Return type List[str]

Returns A list of (unique) source_apis found in the set of statements.

stmts_to_matrix(*stmts*, *extra_evidence=None*)

Convert a list of Statements to a feature matrix.

Features are encoded as follows:

- One column for every source listed in *self.source_list*, containing the number of statement evidences from that source. If *self.include_more_specific* is True and *extra_evidence* is provided, these are used in combination with the Statement's own evidence in determining source counts.
- If *self.use_stmt_type* is set, statement type is included via one-hot encoding, with one column for each statement type.
- If *self.use_num_members* is set, a column is added for the number of agents in the Statement.
- If *self.use_num_pmids* is set, a column is added with the total total number of unique PMIDs supporting the Statement. If *extra_evidence* is provided, these are used in combination with the Statement's own evidence in determining the number of PMIDs.

Parameters

- **stmts** (Sequence[*Statement*]) – A list or tuple of INDRA Statements to be used to generate a feature matrix.
- **extra_evidence** (Optional[List[List[*Evidence*]]]) – A list corresponding to the given list of statements, where each entry is a list of Evidence objects providing additional support for the corresponding statement (i.e., Evidences that aren't already included in the Statement's own evidence list).

Return type ndarray

Returns Feature matrix for the statement data.

class indra.belief.skl.**HybridScorer**(*counts_scorer*, *simple_scorer*)

Use CountsScorer for known sources, SimpleScorer priors for any others.

Allows the use of a CountsScorer to make belief predictions based on sources seen in training data, while falling back to SimpleScorer priors for any sources not accounted for by the CountsScorer. Like the SimpleScorer, uses an independence assumption to combine beliefs from the two scorers (i.e., *hybrid_bel = 1 - (1 - cs_bel) * (1 - ss_bel)*).

Parameters

- **counts_scorer** (*CountsScorer*) – Instance of CountsScorer.
- **simple_scorer** (*SimpleScorer*) – Instance of SimpleScorer.

check_prior_probs(*statements*)

Check that sources in the set of statements are accounted for.

Return type None

score_statements(*statements*, *extra_evidence=None*)

Parameters

- **statements** (Sequence[*Statement*]) – INDRA Statements whose belief scores are to be calculated.
- **extra_evidence** (Optional[List[List[*Evidence*]]]) – A list corresponding to the given list of statements, where each entry is a list of Evidence objects providing additional support for the corresponding statement (i.e., Evidences that aren't already included in the Statement's own evidence list).

Return type Sequence[float]

Returns The computed probabilities for each statement.

class indra.belief.skl.**SklearnScorer**(*model*)

Use a pre-trained Sklearn classifier to predict belief scores.

An implementing instance of this base class has two personalities: as a subclass of BeliefScorer, it implements the functions required by the BeliefEngine, *score_statements* and *check_prior_probs*. It also behaves like an sklearn model by composition, implementing methods *fit*, *predict*, *predict_proba*, and *predict_log_proba*, which are passed through to an internal sklearn model.

A key role of this wrapper class is to implement the preprocessing of statement properties into a feature matrix in a standard way, so that a classifier trained on one corpus of statement data will still work when used on another corpus.

Implementing subclasses must implement at least one of the methods for building the feature matrix, *stmts_to_matrix* or *df_to_matrix*.

Parameters **model** (BaseEstimator) – Any instance of a classifier object supporting the methods *fit*, *predict_proba*, *predict*, and *predict_log_proba*.

check_prior_probs(*statements*)

Empty implementation for now.

Return type None

df_to_matrix(*df*)

Convert a statement DataFrame to a feature matrix.

Return type ndarray

fit(*stmt_data*, *y_arr*, *extra_evidence=None*, **args*, ***kwargs*)

Preprocess stmt data and run sklearn model *fit* method.

Additional *args* and *kwargs* are passed to the *fit* method of the wrapped sklearn model.

Parameters

- **stmt_data** (Union[ndarray, Sequence[*Statement*], DataFrame]) – Statement content to be used to generate a feature matrix.
- **y_arr** (Sequence[float]) – Class values for the statements (e.g., a vector of 0s and 1s indicating correct or incorrect).
- **extra_evidence** (Optional[List[List[*Evidence*]]]) – A list corresponding to the given list of statements, where each entry is a list of Evidence objects providing additional support for the corresponding statement (i.e., Evidences that aren't already included in the Statement's own evidence list).

predict(*stmt_data*, *extra_evidence=None*, **args*, ***kwargs*)

Preprocess stmt data and run sklearn model *predict* method.

Additional *args* and *kwargs* are passed to the *predict* method of the wrapped sklearn model.

Parameters

- **stmt_data** (Union[ndarray, Sequence[*Statement*], DataFrame]) – Statement content to be used to generate a feature matrix.
- **extra_evidence** (Optional[List[List[*Evidence*]]]) – A list corresponding to the given list of statements, where each entry is a list of Evidence objects providing additional support for the corresponding statement (i.e., Evidences that aren't already included in the Statement's own evidence list).

Return type ndarray

predict_log_proba(*stmt_data*, *extra_evidence=None*, **args*, ***kwargs*)
Preprocess stmt data and run sklearn model *predict_log_proba*.

Additional *args* and *kwargs* are passed to the *predict* method of the wrapped sklearn model.

Parameters

- **stmt_data** (Union[ndarray, Sequence[*Statement*], DataFrame]) – Statement content to be used to generate a feature matrix.
- **extra_evidence** (Optional[List[List[*Evidence*]]]) – A list corresponding to the given list of statements, where each entry is a list of Evidence objects providing additional support for the corresponding statement (i.e., Evidences that aren't already included in the Statement's own evidence list).

Return type ndarray

predict_proba(*stmt_data*, *extra_evidence=None*, **args*, ***kwargs*)
Preprocess stmt data and run sklearn model *predict_proba* method.

Additional *args* and *kwargs* are passed to the *predict_proba* method of the wrapped sklearn model.

Parameters

- **stmt_data** (Union[ndarray, Sequence[*Statement*], DataFrame]) – Statement content to be used to generate a feature matrix.
- **extra_evidence** (Optional[List[List[*Evidence*]]]) – A list corresponding to the given list of statements, where each entry is a list of Evidence objects providing additional support for the corresponding statement (i.e., Evidences that aren't already included in the Statement's own evidence list).

Return type ndarray

score_statements(*statements*, *extra_evidence=None*)
Computes belief probabilities for a list of INDRA Statements.

The Statements are assumed to be de-duplicated. In other words, each Statement is assumed to have a list of Evidence objects that supports it. The probability of correctness of the Statement is generally calculated based on the number of Evidences it has, their sources, and other features depending on the subclass implementation.

Parameters

- **statements** (Sequence[*Statement*]) – INDRA Statements whose belief scores are to be calculated.
- **extra_evidence** (Optional[List[List[*Evidence*]]]) – A list corresponding to the given list of statements, where each entry is a list of Evidence objects providing additional support for the corresponding statement (i.e., Evidences that aren't already included in the Statement's own evidence list).

Return type Sequence[float]

Returns The computed prior probabilities for each statement.

stmts_to_matrix(*stmts*, *extra_evidence=None*)
Convert a list of Statements to a feature matrix.

Return type ndarray

to_matrix(*stmt_data*, *extra_evidence=None*)
Get stmt feature matrix by calling appropriate method.

If *stmt_data* is already a matrix (e.g., obtained after performing a train/test split on a matrix generated for a full statement corpus), it is returned directly; if a DataFrame of Statement metadata, *self.df_to_matrix* is called; if a list of Statements, *self.stmts_to_matrix* is called.

Parameters

- **stmt_data** (Union[ndarray, Sequence[*Statement*], DataFrame]) – Statement content to be used to generate a feature matrix.
- **extra_evidence** (Optional[List[List[*Evidence*]]]) – A list corresponding to the given list of statements, where each entry is a list of Evidence objects providing additional support for the corresponding statement (i.e., Evidences that aren't already included in the Statement's own evidence list).

Return type ndarray

Returns Feature matrix for the statement data.

4.8 Mechanism Linker (indra.mechlinker)

class indra.mechlinker.**AgentState**(*agent*, *evidence=None*)

A class representing Agent state without identifying a specific Agent.

bound_conditions

Type list[indra.statements.BoundCondition]

mods

Type list[indra.statements.ModCondition]

mutations

Type list[indra.statements.Mutation]

location

Type indra.statements.location

apply_to(*agent*)

Apply this object's state to an Agent.

Parameters **agent** (*indra.statements.Agent*) – The agent to which the state should be applied

class indra.mechlinker.**BaseAgent**(*name*)

Represents all activity types and active forms of an Agent.

Parameters

- **name** (*str*) – The name of the BaseAgent
- **activity_types** (*list[str]*) – A list of activity types that the Agent has
- **active_states** (*dict*) – A dict of activity types and their associated Agent states
- **activity_reductions** (*dict*) – A dict of activity types and the type they are reduced to by inference.

class indra.mechlinker.**BaseAgentSet**
Container for a set of BaseAgents.

This class wraps a dict of BaseAgent instance and can be used to get and set BaseAgents.

get_create_base_agent(*agent*)
Return BaseAgent from an Agent, creating it if needed.

Parameters **agent** (*indra.statements.Agent*) –

Returns **base_agent**

Return type *indra.mechlinker.BaseAgent*

class indra.mechlinker.**LinkedStatement**(*source_stmts*, *inferred_stmt*)
A tuple containing a list of source Statements and an inferred Statement.

The list of source Statements are the basis for the inferred Statement.

Parameters

- **source_stmts** (*list[indra.statements.Statement]*) – A list of source Statements
- **inferred_stmts** (*indra.statements.Statement*) – A Statement that was inferred from the source Statements.

class indra.mechlinker.**MechLinker**(*stmts=None*)
Rewrite the activation pattern of Statements and derive new Statements.

The mechanism linker (MechLinker) traverses a corpus of Statements and uses various inference steps to make the activity types and active forms consistent among Statements.

add_statements(*stmts*)
Add statements to the MechLinker.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of Statements to add.

gather_explicit_activities()
Aggregate all explicit activities and active forms of Agents.

This function iterates over self.statements and extracts explicitly stated activity types and active forms for Agents.

gather_implicit_activities()
Aggregate all implicit activities and active forms of Agents.

Iterate over self.statements and collect the implied activities and active forms of Agents that appear in the Statements.

Note that using this function to collect implied Agent activities can be risky. Assume, for instance, that a Statement from a reading system states that EGF bound to EGFR phosphorylates ERK. This would be interpreted as implicit evidence for the EGFR-bound form of EGF to have 'kinase' activity, which is clearly incorrect.

In contrast the alternative pair of this function: gather_explicit_activities collects only explicitly stated activities.

static infer_activations(*stmts*)
Return inferred RegulateActivity from Modification + ActiveForm.

This function looks for combinations of Modification and ActiveForm Statements and infers Activation/Inhibition Statements from them. For example, if we know that A phosphorylates B, and the phosphorylated form of B is active, then we can infer that A activates B. This can also be viewed as having "explained" a given Activation/Inhibition Statement with a combination of more mechanistic Modification + ActiveForm Statements.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of Statements to infer RegulateActivity from.

Returns **linked_stmts** – A list of LinkedStatements representing the inferred Statements.

Return type list[*indra.mechlinker.LinkedStatement*]

static infer_active_forms(*stmts*)
Return inferred ActiveForm from RegulateActivity + Modification.

This function looks for combinations of Activation/Inhibition Statements and Modification Statements, and infers an ActiveForm from them. For example, if we know that A activates B and A phosphorylates B, then we can infer that the phosphorylated form of B is active.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of Statements to infer ActiveForms from.

Returns **linked_stmts** – A list of LinkedStatements representing the inferred Statements.

Return type list[*indra.mechlinker.LinkedStatement*]

static infer_complexes(*stmts*)
Return inferred Complex from Statements implying physical interaction.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of Statements to infer Complexes from.

Returns **linked_stmts** – A list of LinkedStatements representing the inferred Statements.

Return type list[*indra.mechlinker.LinkedStatement*]

static infer_modifications(*stmts*)
Return inferred Modification from RegulateActivity + ActiveForm.

This function looks for combinations of Activation/Inhibition Statements and ActiveForm Statements that imply a Modification Statement. For example, if we know that A activates B, and phosphorylated B is active, then we can infer that A leads to the phosphorylation of B. An additional requirement when making this assumption is that the activity of B should only be dependent on the modified state and not other context - otherwise the inferred Modification is not necessarily warranted.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of Statements to infer Modifications from.

Returns **linked_stmts** – A list of LinkedStatements representing the inferred Statements.

Return type list[*indra.mechlinker.LinkedStatement*]

reduce_activities()
Rewrite the activity types referenced in Statements for consistency.

Activity types are reduced to the most specific form whenever possible. For instance, if 'kinase' is the only specific activity type known for the BaseAgent of BRAF, its generic 'activity' forms are rewritten to 'kinase'.

replace_activations(*linked_stmts=None*)
Remove RegulateActivity Statements that can be inferred out.

This function iterates over self.statements and looks for RegulateActivity Statements that either match or are refined by inferred RegulateActivity Statements that were linked (provided as the linked_stmts argument). It removes RegulateActivity Statements from self.statements that can be explained by the linked statements.

Parameters **linked_stmts** (*Optional[list[indra.mechlinker.LinkedStatement]]*) – A list of linked statements, optionally passed from outside. If None is passed, the

MechLinker runs self.infer_activations to infer RegulateActivities and obtain a list of LinkedStatements that are then used for removing existing Complexes in self.statements.

replace_complexes(*linked_stmts=None*)
Remove Complex Statements that can be inferred out.

This function iterates over self.statements and looks for Complex Statements that either match or are refined by inferred Complex Statements that were linked (provided as the linked_stmts argument). It removes Complex Statements from self.statements that can be explained by the linked statements.

Parameters **linked_stmts** (*Optional[list[indra.mechlinker.LinkedStatement]]*) – A list of linked statements, optionally passed from outside. If None is passed, the MechLinker runs self.infer_complexes to infer Complexes and obtain a list of LinkedStatements that are then used for removing existing Complexes in self.statements.

require_active_forms()
Rewrites Statements with Agents' active forms in active positions.

As an example, the enzyme in a Modification Statement can be expected to be in an active state. Similarly, subjects of RegulateAmount and RegulateActivity Statements can be expected to be in an active form. This function takes the collected active states of Agents in their corresponding BaseAgents and then rewrites other Statements to apply the active Agent states to them.

Returns **new_stmts** – A list of Statements which includes the newly rewritten Statements. This list is also set as the internal Statement list of the MechLinker.

Return type list[indra.statements.Statement]

4.9 Assemblers of model output (indra.assemblers)

4.9.1 Executable PySB models (indra.assemblers.pysb.assembler)

PySB Assembler (indra.assemblers.pysb.assembler)

class indra.assemblers.pysb.assembler.**Param**(*name*, *value*, *unique=False*)
Represent a parameter as an input to the assembly process.

name
The name of the parameter

Type str

value
The value of the parameter

Type float

unique
If True, a suffix is added to the end of the parameter name upon assembly to make sure the parameter is unique in the model. If False, the name attribute is used as is. Default: False

Type Optional[bool]

class indra.assemblers.pysb.assembler.**Policy**(*name*, *parameters=None*, *sites=None*)
Represent a policy that can be associated with a speficic Statement.

name
The name of the policy, e.g. one_step

Type str

parameters
A dict of parameters where each key identifies the role of the parameter with respect to the policy, e.g. 'Km', and the value is a Param object.

Type dict[str, *Param*]

sites
A dict of site names corresponding to the interactions induced by the policy.

Type dict

class indra.assemblers.pysb.assembler.**PysbAssembler**(*statements=None*)
Assembler creating a PySB model from a set of INDRA Statements.

Parameters **statements** (*list[indra.statements.Statement]*) – A list of INDRA Statements to be assembled.

policies
A dictionary of policies that defines assembly policies for Statement types. It is assigned in the constructor.

Type dict

statements
A list of INDRA statements to be assembled.

Type list[indra.statements.Statement]

model
A PySB model object that is assembled by this class.

Type pysb.Model

agent_set
A set of BaseAgents used during the assembly process.

Type *BaseAgentSet*

add_default_initial_conditions(*value=None*)
Set default initial conditions in the PySB model.

Parameters **value** (*Optional[float]*) – Optionally a value can be supplied which will be the initial amount applied. Otherwise a built-in default is used.

add_statements(*stmts*)
Add INDRA Statements to the assembler's list of statements.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of indra.statements.Statement to be added to the statement list of the assembler.

export_model(*format*, *file_name=None*)
Save the assembled model in a modeling formalism other than PySB.

For more details on exporting PySB models, see http://pysb.readthedocs.io/en/latest/modules/export/index.html

Parameters

- **format** (*str*) – The format to export into, for instance "kappa", "bngl", "sbml", "matlab", "mathematica", "potterswheel". See http://pysb.readthedocs.io/en/latest/modules/export/index.html for a list of supported formats. In addition to the formats supported by PySB itself, this method also provides "sbgn" output.
- **file_name** (*Optional[str]*) – An optional file name to save the exported model into.

Returns **exp_str** – The exported model string or object

Return type str or object

make_model(*policies=None*, *initial_conditions=True*, *reverse_effects=False*, *model_name='indra_model'*)
Assemble the PySB model from the collected INDRA Statements.

This method assembles a PySB model from the set of INDRA Statements. The assembled model is both returned and set as the assembler's model argument.

Parameters

- **policies** (*Optional[Union[str, dict]]*) – A string or dictionary that defines one or more assembly policies.

  If policies is a string, it defines a global assembly policy that applies to all Statement types. Example: one_step, interactions_only

  A dictionary of policies has keys corresponding to Statement types and values to the policy to be applied to that type of Statement. For Statement types whose policy is undefined, the 'default' policy is applied. Example: {'Phosphorylation': 'two_step'}

- **initial_conditions** (*Optional[bool]*) – If True, default initial conditions are generated for the Monomers in the model. Default: True

- **reverse_effects** (*Optional[bool]*) – If True, reverse rules are added to the model for activity, modification and amount regulations that have no corresponding reverse effects. Default: False

- **model_name** (*Optional[str]*) – The name attribute assigned to the PySB Model object. Default: "indra_model"

Returns **model** – The assembled PySB model object.

Return type pysb.Model

print_model()
Print the assembled model as a PySB program string.

This function is useful when the model needs to be passed as a string to another component.

save_model(*file_name='pysb_model.py'*)
Save the assembled model as a PySB program file.

Parameters **file_name** (*Optional[str]*) – The name of the file to save the model program code in. Default: pysb-model.py

save_rst(*file_name='pysb_model.rst'*, *module_name='pysb_module'*)
Save the assembled model as an RST file for literate modeling.

Parameters

- **file_name** (*Optional[str]*) – The name of the file to save the RST in. Default: pysb_model.rst

- **module_name** (*Optional[str]*) – The name of the python function defining the module. Default: pysb_module

set_context(*cell_type*)
Set protein expression amounts from CCLE as initial conditions.

This method uses *indra.databases.context_client* to get protein expression levels for a given cell type and set initial conditions for Monomers in the model accordingly.

Parameters

- **cell_type** (*str*) – Cell type name for which expression levels are queried. The cell type name follows the CCLE database conventions.
- **Example** (*LOXIMVI_SKIN, BT20_BREAST*) –

set_expression(*expression_dict*)
Set protein expression amounts as initial conditions

Parameters **expression_dict** (*dict*) – A dictionary in which the keys are gene names and the values are numbers representing the absolute amount (count per cell) of proteins expressed. Proteins that are not expressed can be represented as nan. Entries that are not in the dict or are in there but resolve to None, are set to the default initial amount. Example: {'EGFR': 12345, 'BRAF': 4567, 'ESR1': nan}

exception indra.assemblers.pysb.assembler.**UnknownPolicyError**

indra.assemblers.pysb.assembler.**add_rule_to_model**(*model*, *rule*, *annotations=None*)
Add a Rule to a PySB model and handle duplicate component errors.

indra.assemblers.pysb.assembler.**complex_monomers_default**(*stmt*, *agent_set*)
In this (very simple) implementation, proteins in a complex are each given site names corresponding to each of the other members of the complex (lower case). So the resulting complex can be "fully connected" in that each member can be bound to all the others.

indra.assemblers.pysb.assembler.**complex_monomers_one_step**(*stmt*, *agent_set*)
In this (very simple) implementation, proteins in a complex are each given site names corresponding to each of the other members of the complex (lower case). So the resulting complex can be "fully connected" in that each member can be bound to all the others.

indra.assemblers.pysb.assembler.**get_agent_rule_str**(*agent*)
Construct a string from an Agent as part of a PySB rule name.

indra.assemblers.pysb.assembler.**get_annotation**(*component*, *db_name*, *db_ref*)
Construct model Annotations for each component.

Annotation formats follow guidelines at https://identifiers.org/.

indra.assemblers.pysb.assembler.**get_create_parameter**(*model*, *param*)
Return parameter with given name, creating it if needed.

If unique is false and the parameter exists, the value is not changed; if it does not exist, it will be created. If unique is true then upon conflict a number is added to the end of the parameter name.

Parameters

- **model** (*pysb.Model*) – The model to add the parameter to
- **param** (Param) – An assembly parameter object

indra.assemblers.pysb.assembler.**get_grounded_agents**(*model*)
Given a PySB model, get mappings from rule to monomer patterns and from monomer patterns to grounded agents.

indra.assemblers.pysb.assembler.**get_monomer_pattern**(*model*, *agent*, *extra_fields=None*)
Construct a PySB MonomerPattern from an Agent.

indra.assemblers.pysb.assembler.**get_site_pattern**(*agent*)
Construct a dictionary of Monomer site states from an Agent.

This crates the mapping to the associated PySB monomer from an INDRA Agent object.

indra.assemblers.pysb.assembler.**get_uncond_agent**(*agent*)
Construct the unconditional state of an Agent.

The unconditional Agent is a copy of the original agent but without any bound conditions and modification conditions. Mutation conditions, however, are preserved since they are static.

indra.assemblers.pysb.assembler.**grounded_monomer_patterns**(*model*, *agent*, *ignore_activities=False*)
Get monomer patterns for the agent accounting for grounding information.

Parameters

- **model** (*pysb.core.Model*) – The model to search for MonomerPatterns matching the given Agent.
- **agent** (*indra.statements.Agent*) – The Agent to find matching MonomerPatterns for.
- **ignore_activites** (*bool*) – Whether to ignore any ActivityConditions on the agent when determining the required site conditions for the MonomerPattern. For example, if set to True, will find a match for the agent *MAPK1(activity=kinase)* even if the corresponding MAPK1 Monomer in the model has no site named *kinase*. Default is False (more stringent matching).

Return type generator of MonomerPatterns

indra.assemblers.pysb.assembler.**parse_identifiers_url**(*url*)
Parse an identifiers.org URL into (namespace, ID) tuple.

indra.assemblers.pysb.assembler.**set_base_initial_condition**(*model*, *monomer*, *value*)
Set an initial condition for a monomer in its 'default' state.

indra.assemblers.pysb.assembler.**set_extended_initial_condition**(*model*, *monomer=None*, *value=0*)
Set an initial condition for monomers in "modified" state.

This is useful when using downstream analysis that relies on reactions being active in the model. One example is BioNetGen-based reaction network diagram generation.

PySB PreAssembler (indra.assemblers.pysb.preassembler)

class indra.assemblers.pysb.preassembler.**PysbPreassembler**(*stmts=None*)
Pre-assemble Statements in preparation for PySB assembly.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of Statements to assemble

add_reverse_effects()
Add Statements for the reverse effects of some Statements.

For instance, if a protein is phosphorylated but never dephosphorylated in the model, we add a generic dephosphorylation here. This step is usually optional in the assembly process.

add_statements(*stmts*)
Add a list of Statements for assembly.

replace_activities()
Replace ative flags with Agent states when possible.

Base Agents (indra.assemblers.pysb.base_agents)

class indra.assemblers.pysb.base_agents.**BaseAgent**(*name*)
A BaseAgent aggregates the global properties of an Agent.

The BaseAgent class aggregates the name, sites, site states, active forms, inactive forms and database references of Agents from individual INDRA Statements. This allows the PySB Assembler to correctly assemble the Monomer signatures in the model.

add_activity_form(*activity_pattern*, *is_active*)
Adds the pattern as an active or inactive form to an Agent.

Parameters

- **activity_pattern** (*dict*) – A dictionary of site names and their states.
- **is_active** (*bool*) – Is True if the given pattern corresponds to an active state.

add_activity_type(*activity_type*)
Adds an activity type to an Agent.

Parameters **activity_type** (*str*) – The type of activity to add such as 'activity', 'kinase', 'gtpbound'

add_site_states(*site*, *states*)
Create new states on an agent site if the state doesn't exist.

create_mod_site(*mc*)
Create modification site for the BaseAgent from a ModCondition.

create_site(*site*, *states=None*)
Create a new site on an agent if it doesn't already exist.

class indra.assemblers.pysb.base_agents.**BaseAgentSet**
Container for a dict of BaseAgents with their names as keys.

get_create_base_agent(*agent*)
Return base agent with given name, creating it if needed.

items()
Return items for the set of BaseAgents that this class wraps.

A utility to get graphs from kappa (indra.assemblers.pysb.kappa_util)

indra.assemblers.pysb.kappa_util.**cm_json_to_graph**(*cm_json*)
Return pygraphviz Agraph from Kappy's contact map JSON.

Parameters **cm_json** (*dict*) – A JSON dict which contains a contact map generated by Kappy.

Returns **graph** – A graph representing the contact map.

Return type pygraphviz.Agraph

indra.assemblers.pysb.kappa_util.**cm_json_to_networkx**(*cm_json*)
Return a networkx graph from Kappy's contact map JSON.

The networkx Graph's structure is as follows. Each monomer is represented as a node of type "agent", and each site is represented as a separate node of type "site". Edges that have type "link" connect site nodes whereas edges with type "part" connect monomers with their sites.

Parameters **cm_json** (*dict*) – A JSON dict which contains a contact map generated by Kappy.

Returns **graph** – An undirected graph representing the contact map.

Return type networkx.Graph

indra.assemblers.pysb.kappa_util.**get_cm_cycles**(*cm_graph*)
Return cycles from a model's Kappa contact map graph representation.

Parameters **cm_graph** (*networkx.Graph*) – A networkx graph produced by cm_json_to_networkx.

Returns A list of base cycles found in the contact map graph. Each cycle is represented as a list of strings of the form Monomer(site).

Return type list

indra.assemblers.pysb.kappa_util.**im_json_to_graph**(*im_json*)
Return networkx graph from Kappy's influence map JSON.

Parameters **im_json** (*dict*) – A JSON dict which contains an influence map generated by Kappy.

Returns **graph** – A graph representing the influence map.

Return type networkx.MultiDiGraph

4.9.2 Cytoscape networks (indra.assemblers.cx.assembler)

class indra.assemblers.cx.assembler.**CxAssembler**(*stmts=None*, *network_name=None*)
This class assembles a CX network from a set of INDRA Statements.

The CX format is an aspect oriented data mode for networks. The format is defined at http://www.home.ndexbio.org/data-model/. The CX format is the standard for NDEx and is compatible with CytoScape via the CyNDEx plugin.

Parameters

- **stmts** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to be assembled.
- **network_name** (*Optional[str]*) – The name of the network to be assembled. Default: indra_assembled

statements
A list of INDRA Statements to be assembled.

Type list[indra.statements.Statement]

network_name
The name of the network to be assembled.

Type str

cx
The structure of the CX network that is assembled.

Type dict

add_statements(*stmts*)
Add INDRA Statements to the assembler's list of statements.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of indra.statements.Statement to be added to the statement list of the assembler.

make_model(*add_indra_json=True*)
Assemble the CX network from the collected INDRA Statements.

This method assembles a CX network from the set of INDRA Statements. The assembled network is set as the assembler's cx argument.

Parameters **add_indra_json** (*Optional[bool]*) – If True, the INDRA Statement JSON annotation is added to each edge in the network. Default: True

Returns **cx_str** – The json serialized CX model.

Return type str

print_cx(*pretty=True*)
Return the assembled CX network as a json string.

Parameters **pretty** (*bool*) – If True, the CX string is formatted with indentation (for human viewing) otherwise no indentation is used.

Returns **json_str** – A json formatted string representation of the CX network.

Return type str

save_model(*file_name='model.cx'*)
Save the assembled CX network in a file.

Parameters **file_name** (*Optional[str]*) – The name of the file to save the CX network to. Default: model.cx

set_context(*cell_type*)
Set protein expression data and mutational status as node attribute

This method uses *indra.databases.context_client* to get protein expression levels and mutational status for a given cell type and set a node attribute for proteins accordingly.

Parameters **cell_type** (*str*) – Cell type name for which expression levels are queried. The cell type name follows the CCLE database conventions. Example: LOXIMVI_SKIN, BT20_BREAST

upload_model(*ndex_cred=None*, *private=True*, *style='default'*)
Creates a new NDEx network of the assembled CX model.

To upload the assembled CX model to NDEx, you need to have a registered account on NDEx (http://ndexbio.org/) and have the *ndex* python package installed. The uploaded network is private by default.

Parameters

- **ndex_cred** (*Optional[dict]*) – A dictionary with the following entries: 'user': NDEx user name 'password': NDEx password
- **private** (*Optional[bool]*) – Whether or not the created network will be private on NDEX.
- **style** (*Optional[str]*) – This optional parameter can either be (1) The UUID of an existing NDEx network whose style should be applied to the new network. (2) Unspecified or 'default' to use the default INDRA-assembled network style. (3) None to not set a network style.

Returns **network_id** – The UUID of the NDEx network that was created by uploading the assembled CX model.

Return type str

class indra.assemblers.cx.assembler.**NiceCxAssembler**(*stmts=None*, *network_name=None*)
Assembles a Nice CX network from a set of INDRA Statements.

Parameters

- **stmts** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to be assembled.
- **network_name** (*Optional[str]*) – The name of the network to be assembled. Default: indra_assembled

network
A Nice CX network object that is assembled from Statements.

Type ndex2.nice_cx_network.NiceCXNetwork

add_edge(*a1_id*, *a2_id*, *stmt*)
Add a Statement to the network as an edge.

add_node(*agent*)
Add an Agent to the network as a node.

make_model(*self_loops=False*, *network_attributes=None*)
Return a Nice CX network object after running assembly.

Parameters

- **self_loops** (*Optional[bool]*) – If False, self-loops are excluded from the network. Default: False
- **network_attributes** (*Optional[dict]*) – A dictionary containing attributes to be added to the assembled network.

Returns The assembled Nice CX network.

Return type ndex2.nice_cx_network.NiceCXNetwork

print_model()
Return the CX string of the assembled model.

4.9.3 Natural language (indra.assemblers.english.assembler)

class indra.assemblers.english.assembler.**AgentWithCoordinates**(*agent_str*, *name*, *db_refs*, *coords=None*)
English representation of an agent.

Parameters

- **agent_str** (*str*) – Full English description of an agent.
- **name** (*str*) – Name of an agent.
- **db_refs** (*dict*) – Dictionary of database identifiers associated with this agent.
- **coords** (*tuple(int)*) – A tuple of integers representing coordinates of agent name in a text. If not provided, coords will be set to coordinates of name in agent_str. When AgentWithCoordinates is a part of SentenceBuilder or EnglishAssembler, the coords represent the location of agent name in the SentenceBuilder.sentence or EnglishAssembler.model.

update_coords(*shift_by*)
Update coordinates by shifting them by a given number of characters.

Parameters **shift_by** (*int*) – How many characters to shift the parameters by.

class indra.assemblers.english.assembler.**EnglishAssembler**(*stmts=None*)
This assembler generates English sentences from INDRA Statements.

Parameters **stmts** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to be added to the assembler.

statements
A list of INDRA Statements to assemble.

Type list[indra.statements.Statement]

model
The assembled sentences as a single string.

Type str

stmt_agents
A list containing lists of AgentWithCoordinates objects for each of the assembled statements. Coordinates represent the location of agents in the model.

Type list[list[*AgentWithCoordinates*]]

add_statements(*stmts*)
Add INDRA Statements to the assembler's list of statements.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of `indra.statements.Statement` to be added to the statement list of the assembler.

make_model()
Assemble text from the set of collected INDRA Statements.

Returns **stmt_strs** – Return the assembled text as unicode string. By default, the text is a single string consisting of one or more sentences with periods at the end.

Return type str

class indra.assemblers.english.assembler.**SentenceBuilder**
Builds a sentence from agents and strings.

agents
A list of AgentWithCoordinates objects that are part of a sentence. The coordinates of the agent name are being dynamically updated as the sentence is being constructed.

Type list[*AgentWithCoordinates*]

sentence
A sentence that is being built by the builder.

Type str

append(*element*)
Append an element to the end of the sentence.

Parameters **element** (*str or* `AgentWithCoordinates`) – A string or AgentWithCoordinates object to be appended in the end of the sentence. Agent's name coordinates are updated relative to the current length of the sentence.

append_as_list(*lst*, *oxford=True*)
Append a list of elements in a gramatically correct way.

Parameters

- **lst** (*list[str] or list[*`AgentWithCoordinates`*]*) – A list of elements to append. Elements in this list represent a sequence and grammar standards require the use of appropriate punctuation and conjunction to connect them (e.g. [ag1, ag2, ag3]).
- **oxford** (*Optional[bool]*) – Whether to use oxford grammar standards. Default: True

append_as_sentence(*lst*)
Append a list of elements by concatenating them together.

Note: a list of elements here are parts of sentence that do not represent a sequence and do not need to have extra punctuation or conjunction between them.

Parameters **lst** (*list[str] or list[AgentWithCoordinates]*) – A list of elements to append. Elements in this list do not represent a sequence and do not need to have extra punctuation or conjunction between them (e.g. [subj, ' is a GAP for ', obj]).

make_sentence()
After the parts of a sentence are joined, create a sentence.

prepend(*element*)
Prepend an element to the beginning of the sentence.

Parameters **element** (*str or AgentWithCoordinates*) – A string or AgentWithCoordinates object to be added in the beginning of the sentence. All existing agents' names coordinates are updated relative to the new length of the sentence.

indra.assemblers.english.assembler.**english_join**(*lst*)
Join a list of strings according to English grammar.

Parameters **lst** (*list of str*) – A list of strings to join.

Returns A string which describes the list of elements, e.g., "apples, pears, and bananas".

Return type str

indra.assemblers.english.assembler.**statement_base_verb**(*stmt_type*)
Return the base verb form of a statement type.

Parameters **stmt_type** (*str*) – The lower case string form of a statement type, for instance, 'phosphorylation'.

Returns The base verb form of a statement type, for instance, 'phosphorylate'.

Return type str

indra.assemblers.english.assembler.**statement_passive_verb**(*stmt_type*)
Return the passive / state verb form of a statement type.

Parameters **stmt_type** (*str*) – The lower case string form of a statement type, for instance, 'phosphorylation'.

Returns The passive/state verb form of a statement type, for instance, 'phosphorylated'.

Return type str

indra.assemblers.english.assembler.**statement_present_verb**(*stmt_type*)
Return the present verb form of a statement type.

Parameters **stmt_type** (*str*) – The lower case string form of a statement type, for instance, 'phosphorylation'.

Returns The present verb form of a statement type, for instance, 'phosphorylates'.

Return type str

4.9.4 Node-edge graphs (indra.assemblers.graph.assembler)

class indra.assemblers.graph.assembler.**GraphAssembler**(*stmts=None*, *graph_properties=None*, *node_properties=None*, *edge_properties=None*)

The Graph assembler assembles INDRA Statements into a Graphviz node-edge graph.

Parameters

- **stmts** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to be added to the assembler's list of Statements.
- **graph_properties** (*Optional[dict[str: str]]*) – A dictionary of graphviz graph properties overriding the default ones.
- **node_properties** (*Optional[dict[str: str]]*) – A dictionary of graphviz node properties overriding the default ones.
- **edge_properties** (*Optional[dict[str: str]]*) – A dictionary of graphviz edge properties overriding the default ones.

statements
A list of INDRA Statements to be assembled.

Type list[indra.statements.Statement]

graph
A pygraphviz graph that is assembled by this assembler.

Type pygraphviz.AGraph

existing_nodes
The list of nodes (identified by node key tuples) that are already in the graph.

Type list[tuple]

existing_edges
The list of edges (identified by edge key tuples) that are already in the graph.

Type list[tuple]

graph_properties
A dictionary of graphviz graph properties used for assembly.

Type dict[str: str]

node_properties
A dictionary of graphviz node properties used for assembly.

Type dict[str: str]

edge_properties
A dictionary of graphviz edge properties used for assembly. Note that most edge properties are determined based on the type of the edge by the assembler (e.g. color, arrowhead). These settings cannot be directly controlled through the API.

Type dict[str: str]

add_statements(*stmts*)
Add a list of statements to be assembled.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of INDRA Statements to be appended to the assembler's list.

get_string()
Return the assembled graph as a string.

Returns **graph_string** – The assembled graph as a string.

Return type str

make_model()
Assemble the graph from the assembler's list of INDRA Statements.

save_dot(*file_name='graph.dot'*)
Save the graph in a graphviz dot file.

Parameters **file_name** (*Optional[str]*) – The name of the file to save the graph dot string to.

save_pdf(*file_name='graph.pdf'*, *prog='dot'*)
Draw the graph and save as an image or pdf file.

Parameters

- **file_name** (*Optional[str]*) – The name of the file to save the graph as. Default: graph.pdf
- **prog** (*Optional[str]*) – The graphviz program to use for graph layout. Default: dot

4.9.5 SIF / Boolean networks (indra.assemblers.sif.assembler)

class indra.assemblers.sif.assembler.**SifAssembler**(*stmts=None*)
The SIF assembler assembles INDRA Statements into a networkx graph.

This graph can then be exported into SIF (simple interaction format) or a Boolean network.

Parameters **stmts** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to be added to the assembler's list of Statements.

graph
A networkx graph that is assembled by this assembler.

Type networkx.DiGraph

make_model(*use_name_as_key=False*, *include_mods=False*, *include_complexes=False*)
Assemble the graph from the assembler's list of INDRA Statements.

Parameters

- **use_name_as_key** (*boolean*) – If True, uses the name of the agent as the key to the nodes in the network. If False (default) uses the matches_key() of the agent.
- **include_mods** (*boolean*) – If True, adds Modification statements into the graph as directed edges. Default is False.
- **include_complexes** (*boolean*) – If True, creates two edges (in both directions) between all pairs of nodes in Complex statements. Default is False.

print_boolean_net(*out_file=None*)
Return a Boolean network from the assembled graph.

See https://github.com/ialbert/booleannet for details about the format used to encode the Boolean rules.

Parameters **out_file** (*Optional[str]*) – A file name in which the Boolean network is saved.

Returns **full_str** – The string representing the Boolean network.

Return type str

print_loopy(*as_url=True*)
Return

Parameters **out_file** (*Optional[str]*) – A file name in which the Loopy network is saved.

Returns **full_str** – The string representing the Loopy network.

Return type str

print_model(*include_unsigned_edges=False*)
Return a SIF string of the assembled model.

Parameters **include_unsigned_edges** (*bool*) – If True, includes edges with an unknown activating/inactivating relationship (e.g., most PTMs). Default is False.

save_model(*fname*, *include_unsigned_edges=False*)
Save the assembled model's SIF string into a file.

Parameters

- **fname** (*str*) – The name of the file to save the SIF into.
- **include_unsigned_edges** (*bool*) – If True, includes edges with an unknown activating/inactivating relationship (e.g., most PTMs). Default is False.

4.9.6 MITRE "index cards" (indra.assemblers.index_card.assembler)

class indra.assemblers.index_card.assembler.**IndexCardAssembler**(*statements=None*, *pmc_override=None*)
Assembler creating index cards from a set of INDRA Statements.

Parameters

- **statements** (*list*) – A list of INDRA statements to be assembled.
- **pmc_override** (*Optional[str]*) – A PMC ID to assign to the index card.

statements
A list of INDRA statements to be assembled.

Type list

add_statements(*statements*)
Add statements to the assembler.

Parameters **statements** (*list[indra.statement.Statements]*) – The list of Statements to add to the assembler.

make_model()
Assemble statements into index cards.

print_model()
Return the assembled cards as a JSON string.

Returns **cards_json** – The JSON string representing the assembled cards.

Return type str

save_model(*file_name='index_cards.json'*)
Save the assembled cards into a file.

Parameters **file_name** (*Optional[str]*) – The name of the file to save the cards into. Default: index_cards.json

4.9.7 SBGN output (indra.assemblers.sbgn.assembler)

class indra.assemblers.sbgn.assembler.**SBGNAssembler**(*statements=None*)

This class assembles an SBGN model from a set of INDRA Statements.

The Systems Biology Graphical Notation (SBGN) is a widely used graphical notation standard for systems biology models. This assembler creates SBGN models following the Process Desctiption (PD) standard, documented at: https://github.com/sbgn/process-descriptions/blob/master/UserManual/sbgn_PD-level1-user-public.pdf. For more information on SBGN, see: http://sbgn.github.io/sbgn/

Parameters **stmts** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to be assembled.

statements

A list of INDRA Statements to be assembled.

Type list[indra.statements.Statement]

sbgn

The structure of the SBGN model that is assembled, represented as an XML ElementTree.

Type lxml.etree.ElementTree

add_statements(*stmts*)

Add INDRA Statements to the assembler's list of statements.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of indra.statements.Statement to be added to the statement list of the assembler.

make_model()

Assemble the SBGN model from the collected INDRA Statements.

This method assembles an SBGN model from the set of INDRA Statements. The assembled model is set as the assembler's sbgn attribute (it is represented as an XML ElementTree internally). The model is returned as a serialized XML string.

Returns **sbgn_str** – The XML serialized SBGN model.

Return type str

print_model(*pretty=True*, *encoding='utf8'*)

Return the assembled SBGN model as an XML string.

Parameters **pretty** (*Optional[bool]*) – If True, the SBGN string is formatted with indentation (for human viewing) otherwise no indentation is used. Default: True

Returns **sbgn_str** – An XML string representation of the SBGN model.

Return type bytes (str in Python 2)

save_model(*file_name='model.sbgn'*)

Save the assembled SBGN model in a file.

Parameters **file_name** (*Optional[str]*) – The name of the file to save the SBGN network to. Default: model.sbgn

4.9.8 Cytoscape JS networks (indra.assemblers.cyjs.assembler)

class indra.assemblers.cyjs.assembler.**CyJSAssembler**(*stmts=None*)

This class assembles a CytoscapeJS graph from a set of INDRA Statements.

CytoscapeJS is a web-based network library for analysis and visualisation: http://js.cytoscape.org/

Parameters **statements** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to be assembled.

statements

A list of INDRA Statements to be assembled.

Type list[indra.statements.Statement]

add_statements(*stmts*)

Add INDRA Statements to the assembler's list of statements.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of indra.statements.Statement to be added to the statement list of the assembler.

get_gene_names()

Gather gene names of all nodes and node members

make_model(**args*, ***kwargs*)

Assemble a Cytoscape JS network from INDRA Statements.

This method assembles a Cytoscape JS network from the set of INDRA Statements added to the assembler.

Parameters **grouping** (*bool*) – If True, the nodes with identical incoming and outgoing edges are grouped and the corresponding edges are merged.

Returns **cyjs_str** – The json serialized Cytoscape JS model.

Return type str

print_cyjs_context()

Return a list of node names and their respective context.

Returns **cyjs_str_context** – A json string of the context dictionary. e.g. - {'CCLE' : {'bin_expression' : {'cell_line1' : {'gene1':'val1'} }, 'bin_expression' : {'cell_line' : {'gene1':'val1'} } }}

Return type str

print_cyjs_graph()

Return the assembled Cytoscape JS network as a json string.

Returns **cyjs_str** – A json string representation of the Cytoscape JS network.

Return type str

save_json(*fname_prefix='model'*)

Save the assembled Cytoscape JS network in a json file.

This method saves two files based on the file name prefix given. It saves one json file with the graph itself, and another json file with the context.

Parameters **fname_prefix** (*Optional[str]*) – The prefix of the files to save the Cytoscape JS network and context to. Default: model

save_model(*fname='model.js'*)

Save the assembled Cytoscape JS network in a js file.

Parameters **file_name** (*Optional[str]*) – The name of the file to save the Cytoscape JS network to. Default: model.js

set_CCLE_context(*cell_types*)
Set context of all nodes and node members from CCLE.

4.9.9 Tabular output (indra.assemblers.tsv.assembler)

class indra.assemblers.tsv.assembler.**TsvAssembler**(*statements=None*)
Assembles Statements into a set of tabular files for export or curation.

Currently designed for use with "raw" Statements, i.e., Statements with a single evidence entry. Exports Statements into a single tab-separated file with the following columns:

INDEX A 1-indexed integer identifying the statement.

UUID The UUID of the Statement.

TYPE Statement type, given by the name of the class in indra.statements.

STR String representation of the Statement. Contains most relevant information for curation including any additional statement data beyond the Statement type and Agents.

AG_A_TEXT For Statements extracted from text, the text in the sentence corresponding to the first agent (i.e., the 'TEXT' entry in the db_refs dictionary). For all other Statements, the Agent name is given. Empty field if the Agent is None.

AG_A_LINKS Groundings for the first agent given as a comma-separated list of identifiers.org links. Empty if the Agent is None.

AG_A_STR String representation of the first agent, including additional agent context (e.g. modification, mutation, location, and bound conditions). Empty if the Agent is None.

AG_B_TEXT, AG_B_LINKS, AG_B_STR As above for the second agent. Note that the Agent may be None (and these fields left empty) if the Statement consists only of a single Agent (e.g., SelfModification, ActiveForm, or Translocation statement).

PMID PMID of the first entry in the evidence list for the Statement.

TEXT Evidence text for the Statement.

IS_HYP Whether the Statement represents a "hypothesis", as flagged by some reading systems and recorded in the *evidence.epistemics['hypothesis']* field.

IS_DIRECT Whether the Statement represents a direct physical interactions, as recorded by the *evidence.epistemics['direct']* field.

In addition, if the *add_curation_cols* flag is set when calling *TsvAssembler.make_model()*, the following additional (empty) columns will be added, to be filled out by curators:

AG_A_IDS_CORRECT Correctness of Agent A grounding.

AG_A_STATE_CORRECT Correctness of Agent A context (e.g., modification, bound, and other conditions).

AG_B_IDS_CORRECT, AG_B_STATE_CORRECT As above, for Agent B.

EVENT_CORRECT Whether the event is supported by the evidence text if the entities (Agents A and B) are considered as placeholders (i.e., ignoring the correctness of their grounding).

RES_CORRECT For Modification statements, whether the amino acid residue indicated by the Statement is supported by the evidence.

POS_CORRECT For Modification statements, whether the amino acid position indicated by the Statement is supported by the evidence.

SUBJ_ACT_CORRECT For Activation/Inhibition Statements, whether the activity indicated for the subject (Agent A) is supported by the evidence.

OBJ_ACT_CORRECT For Activation/Inhibition Statements, whether the activity indicated for the object (Agent B) is supported by the evidence.

HYP_CORRECT Whether the Statement is correctly flagged as a hypothesis.

HYP_CORRECT Whether the Statement is correctly flagged as direct.

Parameters **stmts** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to be assembled.

statements
A list of INDRA Statements to be assembled.

Type list[indra.statements.Statement]

make_model(*output_file*, *add_curation_cols=False*, *up_only=False*)
Export the statements into a tab-separated text file.

Parameters

- **output_file** (*str*) – Name of the output file.
- **add_curation_cols** (*bool*) – Whether to add columns to facilitate statement curation. Default is False (no additional columns).
- **up_only** (*bool*) – Whether to include identifiers.org links *only* for the Uniprot grounding of an agent when one is available. Because most spreadsheets allow only a single hyperlink per cell, this can makes it easier to link to Uniprot information pages for curation purposes. Default is False.

4.9.10 HTML browsing and curation (indra.assemblers.html.assembler)

Format a set of INDRA Statements into an HTML-formatted report which also supports curation.

indra.assemblers.html.assembler.**DB_TEXT_COLOR = 'black'**
The text color for database sources when shown as source count badges

indra.assemblers.html.assembler.**READER_TEXT_COLOR = 'white'**
The text color for reader sources when shown as source count badges

indra.assemblers.html.assembler.**generate_source_css**(*fname*, *source_colors=None*)
Save a stylesheet defining color, background-color for the given sources

Parameters

- **fname** (*str*) – Where to save the stylesheet
- **source_colors** (Optional[List[Tuple[str, Dict[str, Union[str, Dict[str, str]]]]]]) – Colors defining the styles. Default: DEFAULT_SOURCE_COLORS.

class indra.assemblers.html.assembler.**HtmlAssembler**(*statements=None*, *summary_metadata=None*, *ev_counts=None*, *beliefs=None*, *source_counts=None*, *curation_dict=None*, *title='INDRA Results'*, *db_rest_url=None*, *sort_by='default'*, *custom_stats=None*, *custom_sources=None*)

Generates an HTML-formatted report from INDRA Statements.

The HTML report format includes statements formatted in English (by the EnglishAssembler), text and metadata for the Evidence object associated with each Statement, and a Javascript-based curation interface linked to the INDRA database (access permitting). The interface allows for curation of statements at the evidence level by letting the user specify type of error and (optionally) provide a short description of of the error.

Parameters

- **statements** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to be added to the assembler. Statements can also be added using the add_statements method after the assembler has been instantiated.
- **summary_metadata** (*Optional[dict]*) – Dictionary of statement corpus metadata such as that provided by the INDRA REST API. Default is None. Each value should be a concise summary of O(1), not of order the length of the list, such as the evidence totals. The keys should be informative human-readable strings. This information is displayed as a tooltip when hovering over the page title.
- **ev_counts** (*Optional[dict]*) – A dictionary of the total evidence available for each statement indexed by hash. If not provided, the statements that are passed to the constructor are used to determine these, with whatever evidences these statements carry.
- **beliefs** (*Optional[dict]*) – A dictionary of the belief of each statement indexed by hash. If not provided, the beliefs of the statements passed to the constructor are used.
- **source_counts** (*Optional[dict]*) – A dictionary of the itemized evidence counts, by source, available for each statement, indexed by hash. If not provided, the statements that are passed to the constructor are used to determine these, with whatever evidences these statements carry.
- **title** (*str*) – The title to be printed at the top of the page.
- **db_rest_url** (*Optional[str]*) – The URL to a DB REST API to use for links out to further evidence. If given, this URL will be prepended to links that load additional evidence for a given Statement. One way to obtain this value is from the configuration entry indra.config.get_config('INDRA_DB_REST_URL'). If None, the URLs are constructed as relative links. Default: None
- **sort_by** (*str or function or None*) – If str, it indicates which parameter to sort by, such as 'belief' or 'ev_count', or 'ag_count'. Those are the default options because they can be derived from a list of statements, however if you give a custom list of stats with the *custom_stats* argument, you may use any of the parameters used to build it. The default, 'default', is mostly a sort by ev_count but also favors statements with fewer agents.

  Alternatively, you may give a function that takes a dict as its single argument, a dictionary of metrics. The contents of this dictionary always include "belief", "ev_count", and "ag_count". If source_counts are given, each source will also be available as an entry (e.g. "reach" and "sparser"). As with string values, you may also add your own custom stats using the *custom_stats* argument.

  The value may also be None, in which case the sort function will return the same value for all elements, and thus the original order of elements will be preserved. This could have

strange effects when statements are grouped (i.e. when *grouping_level* is not 'statement'); such functionality is untested.

- **custom_stats** (*Optional[list]*) – A list of StmtStat objects containing custom statement statistics to be used in sorting of statements and statement groups.
- **custom_sources** (*SourceInfo*) – Use this if the sources in the statements are from sources other than the default ones present in indra/resources/source_info.json The structure of the input must conform to:

  {

  "source_key": { "name": "Source Name", "link": "<url>", "type": "reader|database", "domain": "<domain>", "default_style": {

  "color": "<text color>", "background-color": "<badge color>"

  }

  }

  Where <text color> and <badge color> must be color names or color codes allowed in an html document per the CSS3 specification: https://www.w3.org/TR/css-color-3/#svg-color

statements
A list of INDRA Statements to assemble.

Type list[indra.statements.Statement]

model
The HTML report formatted as a single string.

Type str

metadata
Dictionary of statement list metadata such as that provided by the INDRA REST API.

Type dict

ev_counts
A dictionary of the total evidence available for each statement indexed by hash.

Type dict

beliefs
A dictionary of the belief score of each statement, indexed by hash.

Type dict

db_rest_url
The URL to a DB REST API.

Type str

add_statements(*statements*)
Add a list of Statements to the assembler.

Parameters **statements** (*list[indra.statements.Statement]*) – A list of INDRA Statements to be added to the assembler.

make_json_model(*grouping_level='agent-pair'*, *no_redundancy=False*, ***kwargs*)
Return the JSON used to create the HTML display.

Parameters

- **grouping_level** (*Optional[str]*) – Statements can be grouped at three levels, 'statement' (ungrouped), 'relation' (grouped by agents and type), and 'agent-pair' (grouped by ordered pairs of agents). Default: 'agent-pair'.
- **no_redundancy** (*Optional[bool]*) – If True, any group of statements that was already presented under a previous heading will be skipped. This is typically the case for complexes where different permutations of complex members are presented. By setting this argument to True, these can be eliminated. Default: False

Returns **json** – A complexly structured JSON dict containing grouped statements and various metadata.

Return type dict

make_model(*template=None*, *grouping_level='agent-pair'*, *add_full_text_search_link=False*, *no_redundancy=False*, ***template_kwargs*)

Return the assembled HTML content as a string.

Parameters

- **template** (*a Template object*) – Manually pass a Jinja template to be used in generating the HTML. The template is responsible for rendering essentially the output of *make_json_model*.
- **grouping_level** (*Optional[str]*) – Statements can be grouped under subheadings at three levels, 'statement' (ungrouped), 'relation' (grouped by agents and type), and 'agent-pair' (grouped by ordered pairs of agents). Default: 'agent-pair'.
- **add_full_text_search_link** (*bool*) – If True, link with Text fragment search in PMC journal will be added for the statements.
- **no_redundancy** (*Optional[bool]*) – If True, any group of statements that was already presented under a previous heading will be skipped. This is typically the case for complexes where different permutations of complex members are presented. By setting this argument to True, these can be eliminated. Default: False

  All other keyword arguments are passed along to the template. If you are using a custom template with args that are not passed below, this is how you pass them.

Returns The assembled HTML as a string.

Return type str

append_warning(*msg*)

Append a warning message to the model to expose issues.

save_model(*fname*, ***kwargs*)

Save the assembled HTML into a file.

Other kwargs are passed directly to *make_model*.

Parameters **fname** (*str*) – The path to the file to save the HTML into.

indra.assemblers.html.assembler.**src_url**(*ev*)

Given an Evidence object, provide the URL for the source

Return type str

indra.assemblers.html.assembler.**tag_text**(*text*, *tag_info_list*)

Apply start/end tags to spans of the given text.

Parameters

- **text** (*str*) – Text to be tagged

- **tag_info_list** (*list of tuples*) – Each tuple refers to a span of the given text. Fields are *(start_ix, end_ix, substring, start_tag, close_tag)*, where substring, start_tag, and close_tag are strings. If any of the given spans of text overlap, the longest span is used.

Returns String where the specified substrings have been surrounded by the given start and close tags.

Return type str

4.9.11 BMI wrapper for PySB-assembled models (indra.assemblers.pysb.bmi_wrapper)

This module allows creating a Basic Modeling Interface (BMI) model from and automatically assembled PySB model. The BMI model can be instantiated within a simulation workflow system where it is simulated together with other models.

class indra.assemblers.pysb.bmi_wrapper.**BMIModel**(*model*, *inputs=None*, *stop_time=1000*, *outside_name_map=None*)

This class represents a BMI model wrapping a model assembled by INDRA.

Parameters

- **model** (*pysb.Model*) – A PySB model assembled by INDRA to be wrapped in BMI.
- **inputs** (*Optional[list[str]]*) – A list of variable names that are considered to be inputs to the model meaning that they are read from other models. Note that designating a variable as input means that it must be provided by another component during the simulation.
- **stop_time** (*int*) – The stopping time for this model, controlling the time units up to which the model is simulated.
- **outside_name_map** (*dict*) – A dictionary mapping outside variables names to inside variable names (i.e. ones that are in the wrapped model)

export_into_python()

Write the model into a pickle and create a module that loads it.

The model basically exports itself as a pickle file and a Python file is then written which loads the pickle file. This allows importing the model in the simulation workflow.

finalize()

Finish the simulation and clean up resources as needed.

get_attribute(*att_name*)

Return the value of a given attribute.

Atrributes include: model_name, version, author_name, grid_type, time_step_type, step_method, time_units

Parameters **att_name** (*str*) – The name of the attribute whose value should be returned.

Returns **value** – The value of the attribute

Return type str

get_current_time()

Return the current time point that the model is at during simulation

Returns **time** – The current time point

Return type float

`get_input_var_names()`
Return a list of variables names that can be set as input.

Returns **var_names** – A list of variable names that can be set from the outside

Return type list[str]

`get_output_var_names()`
Return a list of variables names that can be read as output.

Returns **var_names** – A list of variable names that can be read from the outside

Return type list[str]

`get_start_time()`
Return the initial time point of the model.

Returns **start_time** – The initial time point of the model.

Return type float

`get_status()`
Return the current status of the model.

`get_time_step()`
Return the time step associated with model simulation.

Returns **dt** – The time step for model simulation

Return type float

`get_time_units()`
Return the time units of the model simulation.

Returns **units** – The time unit of simulation as a string

Return type str

`get_value(var_name)`
Return the value of a given variable.

Parameters **var_name** (*str*) – The name of the variable whose value should be returned

Returns **value** – The value of the given variable in the current state

Return type float

`get_values(var_name)`
Return the value of a given variable.

Parameters **var_name** (*str*) – The name of the variable whose value should be returned

Returns **value** – The value of the given variable in the current state

Return type float

`get_var_name(var_name)`
Return the internal variable name given an outside variable name.

Parameters **var_name** (*str*) – The name of the outside variable to map

Returns **internal_var_name** – The internal name of the corresponding variable

Return type str

get_var_rank(*var_name*)
Return the matrix rank of the given variable.

Parameters **var_name** (*str*) – The name of the variable whose rank should be returned

Returns **rank** – The dimensionality of the variable, 0 for scalar, 1 for vector, etc.

Return type int

get_var_type(*var_name*)
Return the type of a given variable.

Parameters **var_name** (*str*) – The name of the variable whose type should be returned

Returns **unit** – The type of the variable as a string

Return type str

get_var_units(*var_name*)
Return the units of a given variable.

Parameters **var_name** (*str*) – The name of the variable whose units should be returned

Returns **unit** – The units of the variable

Return type str

initialize(*cfg_file=None*, *mode=None*)
Initialize the model for simulation, possibly given a config file.

Parameters **cfg_file** (*Optional[str]*) – The name of the configuration file to load, optional.

make_repository_component()
Return an XML string representing this BMI in a workflow.

This description is required by EMELI to discover and load models.

Returns **xml** – String serialized XML representation of the component in the model repository.

Return type str

set_value(*var_name*, *value*)
Set the value of a given variable to a given value.

Parameters

- **var_name** (*str*) – The name of the variable in the model whose value should be set.
- **value** (*float*) – The value the variable should be set to

set_values(*var_name*, *value*)
Set the value of a given variable to a given value.

Parameters

- **var_name** (*str*) – The name of the variable in the model whose value should be set.
- **value** (*float*) – The value the variable should be set to

update(*dt=None*)
Simulate the model for a given time interval.

Parameters **dt** (*Optional[float]*) – The time step to simulate, if None, the default built-in time step is used.

4.9.12 PyBEL graphs (indra.assemblers.pybel.assembler)

class indra.assemblers.pybel.assembler.**PybelAssembler**(*stmts=None*, *name=None*, *description=None*, *version=None*, *authors=None*, *contact=None*, *license=None*, *copyright=None*, *disclaimer=None*, *annotations_to_include=None*)

Assembles a PyBEL graph from a set of INDRA Statements.

PyBEL tools can subsequently be used to export the PyBEL graph into BEL script files, SIF files, and other related output formats.

Parameters

- **stmts** (list[indra.statement.Statement]) – The list of Statements to assemble.
- **name** (*str*) – Name of the assembled PyBEL network.
- **description** (*str*) – Description of the assembled PyBEL network.
- **version** (*str*) – Version of the assembled PyBEL network.
- **authors** (*str*) – Author(s) of the network.
- **contact** (*str*) – Contact information (email) of the responsible author.
- **license** (*str*) – License information for the network.
- **copyright** (*str*) – Copyright information for the network.
- **disclaimer** (*str*) – Any disclaimers for the network.
- **annotations_to_include** (*Optional[list[str]]*) – A list of evidence annotation keys that should be added to the assembled PyBEL graph. Default: None

Examples

```
>>> from indra.statements import *
>>> map2k1 = Agent('MAP2K1', db_refs={'HGNC': '6840'})
>>> mapk1 = Agent('MAPK1', db_refs={'HGNC': '6871'})
>>> stmt = Phosphorylation(map2k1, mapk1, 'T', '185')
>>> pba = PybelAssembler([stmt])
>>> belgraph = pba.make_model()
>>> sorted(node.as_bel() for node in belgraph)
['p(HGNC:6840 ! MAP2K1)', 'p(HGNC:6871 ! MAPK1)', 'p(HGNC:6871 ! MAPK1,␣
↪pmod(go:0006468 ! "protein phosphorylation", Thr, 185))']
>>> len(belgraph)
3
>>> belgraph.number_of_edges()
2
```

save_model(*path*, *output_format=None*)

Save the pybel.BELGraph using one of the outputs from pybel

Parameters

- **path** (*str*) – The path to output to
- **output_format** (*Optional[str]*) – Output format as cx, pickle, json or defaults to bel

to_database(*manager=None*)
Send the model to the PyBEL database

This function wraps pybel.to_database().

Parameters **manager** (*Optional[pybel.manager.Manager]*) – A PyBEL database manager. If none, first checks the PyBEL configuration for PYBEL_CONNECTION then checks the environment variable PYBEL_REMOTE_HOST. Finally, defaults to using SQLite database in PyBEL data directory (automatically configured by PyBEL)

Returns **network** – The SQLAlchemy model representing the network that was uploaded. Returns None if upload fails.

Return type Optional[pybel.manager.models.Network]

to_web(*host=None*, *user=None*, *password=None*)
Send the model to BEL Commons by wrapping pybel.to_web()

The parameters host, user, and password all check the PyBEL configuration, which is located at ~/.config/pybel/config.json by default

Parameters

- **host** (*Optional[str]*) – The host name to use. If none, first checks the PyBEL configuration entry PYBEL_REMOTE_HOST, then the environment variable PYBEL_REMOTE_HOST. Finally, defaults to https://bel-commons.scai.fraunhofer.de.
- **user** (*Optional[str]*) – The username (email) to use. If none, first checks the PyBEL configuration entry PYBEL_REMOTE_USER, then the environment variable PYBEL_REMOTE_USER.
- **password** (*Optional[str]*) – The password to use. If none, first checks the PyBEL configuration entry PYBEL_REMOTE_PASSWORD, then the environment variable PYBEL_REMOTE_PASSWORD.

Returns **response** – The response from the BEL Commons network upload endpoint.

Return type requests.Response

indra.assemblers.pybel.assembler.**get_causal_edge**(*stmt*, *activates*)
Returns the causal, polar edge with the correct "contact".

4.9.13 Kami models (indra.assemblers.kami.assembler)

class indra.assemblers.kami.assembler.**KamiAssembler**(*statements=None*)
make_model(*policies=None*, *initial_conditions=True*, *reverse_effects=False*)
Assemble the Kami model from the collected INDRA Statements.

This method assembles a Kami model from the set of INDRA Statements. The assembled model is both returned and set as the assembler's model argument.

Parameters

- **policies** (*Optional[Union[str, dict]]*) – A string or dictionary of policies, as defined in indra.assemblers.KamiAssembler. This set of policies locally supersedes the default setting in the assembler. This is useful when this function is called multiple times with different policies.
- **initial_conditions** (*Optional[bool]*) – If True, default initial conditions are generated for the agents in the model.

Returns **model** – The assembled Kami model.

Return type dict

class indra.assemblers.kami.assembler.**Nugget**(*id*, *name*, *rate*)
Represents a Kami Nugget.

add_agent(*agent*)
Add an INDRA Agent and its conditions to the Nugget.

add_edge(*from_node*, *to_node*)
Add an edge between two nodes to the Nugget.

add_node(*name_base*, *attrs=None*)
Add a node with a given base name to the Nugget and return ID.

add_typing(*node_id*, *typing*)
Add typing information to a node in the Nugget.

get_nugget_dict()
Return the Nugget as a dictionary.

get_typing_dict()
Return the Nugget's typing information as a dictionary.

4.9.14 IndraNet Graphs (indra.assemblers.indranet)

The IndraNet assembler creates multiple different types of networkx graphs from INDRA Statements. It also allows exporting binary Statement information as a pandas DataFrame.

class indra.assemblers.indranet.net.**IndraNet**(*incoming_graph_data=None*, ***attr*)
A Networkx representation of INDRA Statements.

classmethod digraph_from_df(*df*, *flattening_method=None*, *weight_mapping=None*)
Create a digraph from a pandas DataFrame.

Parameters

- **df** (*pd.DataFrame*) – The dataframe to build the graph from.
- **flattening_method** (*str or function(networkx.DiGraph, edge)*) – The method to use when updating the belief for the flattened edge.
- **weight_mapping** (*function(networkx.DiGraph)*) – A function taking at least the graph G as an argument and returning G after adding edge weights as an edge attribute to the flattened edges using the reserved keyword 'weight'.

Returns An IndraNet graph flattened to a DiGraph

Return type *IndraNet*(nx.DiGraph)

classmethod from_df(*df*)
Create an IndraNet MultiDiGraph from a pandas DataFrame.

Returns an instance of IndraNet with graph data filled out from a dataframe containing pairwise interactions.

Parameters **df** (*pd.DataFrame*) – A `pandas.DataFrame` with each row containing node and edge data for one edge. Indices are used to distinguish multiedges between a pair of nodes. Any columns not part of the below mentioned mandatory columns are considered extra attributes. Columns starting with 'agA_' or 'agB_' (excluding the agA/B_name) will be added to its respective nodes as node attributes. Any other columns will be added as edge attributes.

Mandatory columns are : *agA_name*, *agB_name*, *agA_ns*, *agA_id*, *agB_ns*, *agB_id*, *stmt_type*, *evidence_count*, *stmt_hash*, *belief* and *source_counts*.

Returns An IndraNet object

Return type *IndraNet*

classmethod signed_from_df(*df*, *sign_dict=None*, *flattening_method=None*, *weight_mapping=None*)
Create a signed graph from a pandas DataFrame.

Parameters

- **df** (*pd.DataFrame*) – The dataframe to build the signed graph from.
- **sign_dict** (*dict*) – A dictionary mapping a Statement type to a sign to be used for the edge. By default only Activation and IncreaseAmount are added as positive edges and Inhibition and DecreaseAmount are added as negative edges, but a user can pass any other Statement types in a dictionary.
- **flattening_method** (*str or function(networkx.DiGraph, edge)*) – The method to use when updating the belief for the flattened edge.
- **weight_mapping** (*function(networkx.DiGraph)*) – A function taking at least the graph G as an argument and returning G after adding edge weights as an edge attribute to the flattened edges using the reserved keyword 'weight'.

Returns An IndraNet graph flattened to a signed graph

Return type *IndraNet*(nx.MultiDiGraph)

to_digraph(*flattening_method=None*, *weight_mapping=None*)
Flatten the IndraNet to a DiGraph

Parameters

- **flattening_method** (*str|function*) – The method to use when updating the belief for the flattened edge
- **weight_mapping** (*function*) – A function taking at least the graph G as an argument and returning G after adding edge weights as an edge attribute to the flattened edges using the reserved keyword 'weight'.

Returns **G** – An IndraNet graph flattened to a DiGraph

Return type *IndraNet*(nx.DiGraph)

to_signed_graph(*sign_dict=None*, *flattening_method=None*, *weight_mapping=None*)
Flatten the IndraNet to a signed graph.

Parameters

- **sign_dict** (*dict*) – A dictionary mapping a Statement type to a sign to be used for the edge. By default only Activation and IncreaseAmount are added as positive edges and Inhibition and DecreaseAmount are added as negative edges, but a user can pass any other Statement types in a dictionary.
- **flattening_method** (*str or function(networkx.DiGraph, edge)*) – The method to use when updating the belief for the flattened edge.

  If a string is provided, it must be one of the predefined options 'simple_scorer' or 'complementary_belief'.

  If a function is provided, it must take the flattened graph 'G' and an edge 'edge' to perform the belief flattening on and return a number:

```
>>> def flattening_function(G, edge):
...     # Return the average belief score of the constituent␣
↪edges
...     all_beliefs = [s['belief']
...         for s in G.edges[edge]['statements']]
...     return sum(all_beliefs)/len(all_beliefs)
```

- **weight_mapping** (*function(networkx.DiGraph)*) – A function taking at least the graph G as an argument and returning G after adding edge weights as an edge attribute to the flattened edges using the reserved keyword 'weight'.

  Example:

```
>>> def weight_mapping(G):
...     # Sets the flattened weight to the average of the
...     # inverse source count
...     for edge in G.edges:
...         w = [1/s['evidence_count']
...             for s in G.edges[edge]['statements']]
...         G.edges[edge]['weight'] = sum(w)/len(w)
...     return G
```

Returns **SG** – An IndraNet graph flattened to a signed graph

Return type *IndraNet*(nx.MultiDiGraph)

class indra.assemblers.indranet.assembler.**IndraNetAssembler**(*statements=None*)

Assembler to create an IndraNet object from a list of INDRA statements.

Parameters **statements** (*list[indra.statements.Statement]*) – A list of INDRA Statements to be assembled.

model

An IndraNet graph object assembled by this class.

Type *IndraNet*

add_statements(*stmts*)

Add INDRA Statements to the assembler's list of statements.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of indra.statements.Statement to be added to the statement list of the assembler.

make_df(*exclude_stmts=None*, *complex_members=3*, *extra_columns=None*, *keep_self_loops=True*)

Create a dataframe containing information extracted from assembler's list of statements necessary to build an IndraNet.

Parameters

- **exclude_stmts** (*list[str]*) – A list of statement type names to not include in the dataframe.
- **complex_members** (*int*) – Maximum allowed size of a complex to be included in the data frame. All complexes larger than complex_members will be rejected. For accepted complexes, all permutations of their members will be added as dataframe records. Default is *3*.
- **extra_columns** (*list[tuple(str, function)]*) – A list of tuples defining columns to add to the dataframe in addition to the required columns. Each tuple contains the column name and a function to generate a value from a statement.

- **keep_self_loops** (*Optional[bool]*) – Whether to keep the self-loops when constructing the graph.

Returns

df – Pandas DataFrame object containing information extracted from statements. It contains the following columns:

agA_name The first Agent's name.

agA_ns The first Agent's identifier namespace as per *db_refs*.

agA_id The first Agent's identifier as per *db_refs*

ags_ns, agB_name, agB_id As above for the second agent. Note that the Agent may be None (and these fields left empty) if the Statement consists only of a single Agent (e.g., SelfModification, ActiveForm, or Translocation statement).

stmt_type Statement type, given by the name of the class in indra.statements.

evidence_count Number of evidences for the statement.

stmt_hash An unique long integer hash identifying the content of the statement.

belief The belief score associated with the statement.

source_counts The number of evidences per input source for the statement.

residue If applicable, the amino acid residue being modified. NaN if if it is unknown or unspecified/not applicable.

position If applicable, the position of the modified amino acid. NaN if it is unknown or unspecified/not applicable.

initial_sign The default sign (polarity) associated with the given statement if the statement type has implied polarity. To facilitate weighted path finding, the sign is represented as 0 for positive polarity and 1 for negative polarity.

More columns can be added by providing the extra_columns parameter.

Return type pd.DataFrame

make_model(*method='preassembly'*, *exclude_stmts=None*, *complex_members=3*, *graph_type='multi_graph'*, *sign_dict=None*, *belief_flattening=None*, *belief_scorer=None*, *weight_flattening=None*, *extra_columns=None*, *keep_self_loops=True*)

Assemble an IndraNet graph object.

Parameters

- **method** (*str*) – Method for assembling an IndraNet graph. Accepted values: *df* and *preassembly*. With the *df* method, the statements are converted into pandas DataFrame first where each row corresponds to an edge in unflattened MultiDiGraph IndraNet. Then the IndraNet can be flattened into signed and unsigned graphs. The beliefs can be calculated on the new edges by providing belief_flattening function. With the *preassembly* option, the statements are merged together and the beliefs are calculated by leveraging the preassembly functionality with custom matches functions (the matches functions are applied depending on the graph type). This method ensures the more robust belief calculation.
- **exclude_stmts** (*list[str]*) – A list of statement type names to not include in the graph.

- **complex_members** (*int*) – Maximum allowed size of a complex to be included in the graph. All complexes larger than complex_members will be rejected. For accepted complexes, all permutations of their members will be added as edges. Default is *3*.
- **graph_type** (*str*) – Specify the type of graph to assemble. Chose from 'multi_graph' (default), 'digraph', 'signed'. Default is *multi_graph*.
- **sign_dict** (*dict*) – A dictionary mapping a Statement type to a sign to be used for the edge. This parameter is only used with the 'signed' option. See IndraNet.to_signed_graph for more info.
- **belief_flattening** (*str or function(networkx.DiGraph, edge)*) – Only needed when method is set to *df*. The method to use when updating the belief for the flattened edge.

  If a string is provided, it must be one of the predefined options 'simple_scorer' or 'complementary_belief'.

  If a function is provided, it must take the flattened graph 'G' and an edge 'edge' to perform the belief flattening on and return a number:

```
>>> def belief_flattening(G, edge):
...     # Return the average belief score of the constituent␣
↪edges
...     all_beliefs = [s['belief']
...         for s in G.edges[edge]['statements']]
...     return sum(all_beliefs)/len(all_beliefs)
```

- **belief_scorer** (*Optional[indra.belief.BeliefScorer]*) – Only needed when method is set to *preassembly*. Instance of BeliefScorer class to use in calculating edge probabilities. If None is provided (default), then the default scorer is used.
- **weight_flattening** (*function(networkx.DiGraph)*) – A function taking at least the graph G as an argument and returning G after adding edge weights as an edge attribute to the flattened edges using the reserved keyword 'weight'.

  Example:

```
>>> def weight_flattening(G):
...     # Sets the flattened weight to the average of the
...     # inverse source count
...     for edge in G.edges:
...         w = [1/s['evidence_count']
...             for s in G.edges[edge]['statements']]
...         G.edges[edge]['weight'] = sum(w)/len(w)
...     return G
```

- **keep_self_loops** (*Optional[bool]*) – Whether to keep the self-loops when constructing the graph.

Returns **model** – IndraNet graph object.

Return type *IndraNet*

make_model_by_preassembly(*exclude_stmts=None*, *complex_members=3*, *graph_type='multi_graph'*, *sign_dict=None*, *belief_scorer=None*, *weight_flattening=None*, *extra_columns=None*, *keep_self_loops=True*)

Assemble an IndraNet graph object by preassembling the statements according to selected graph type.

Parameters

- **exclude_stmts** (*list[str]*) – A list of statement type names to not include in the graph.
- **complex_members** (*int*) – Maximum allowed size of a complex to be included in the graph. All complexes larger than complex_members will be rejected. For accepted complexes, all permutations of their members will be added as edges. Default is *3*.
- **graph_type** (*str*) – Specify the type of graph to assemble. Chose from 'multi_graph' (default), 'digraph', 'signed'. Default is *multi_graph*.
- **sign_dict** (*dict*) – A dictionary mapping a Statement type to a sign to be used for the edge. This parameter is only used with the 'signed' option. See IndraNet.to_signed_graph for more info.
- **belief_scorer** (*Optional[indra.belief.BeliefScorer]*) – Instance of BeliefScorer class to use in calculating edge probabilities. If None is provided (default), then the default scorer is used.
- **weight_flattening** (*function(networkx.DiGraph)*) – A function taking at least the graph G as an argument and returning G after adding edge weights as an edge attribute to the flattened edges using the reserved keyword 'weight'.

  Example:

```
>>> def weight_flattening(G):
...     # Sets the flattened weight to the average of the
...     # inverse source count
...     for edge in G.edges:
...         w = [1/s['evidence_count']
...             for s in G.edges[edge]['statements']]
...         G.edges[edge]['weight'] = sum(w)/len(w)
...     return G
```

- **keep_self_loops** (*Optional[bool]*) – Whether to keep the self-loops when constructing the graph.

Returns **model** – IndraNet graph object.

Return type *IndraNet*

make_model_from_df(*exclude_stmts=None*, *complex_members=3*, *graph_type='multi_graph'*, *sign_dict=None*, *belief_flattening=None*, *weight_flattening=None*, *extra_columns=None*, *keep_self_loops=True*)

Assemble an IndraNet graph object by constructing a pandas Dataframe first.

Parameters

- **exclude_stmts** (*list[str]*) – A list of statement type names to not include in the graph.
- **complex_members** (*int*) – Maximum allowed size of a complex to be included in the graph. All complexes larger than complex_members will be rejected. For accepted complexes, all permutations of their members will be added as edges. Default is *3*.
- **graph_type** (*str*) – Specify the type of graph to assemble. Chose from 'multi_graph' (default), 'digraph', 'signed'. Default is *multi_graph*.
- **sign_dict** (*dict*) – A dictionary mapping a Statement type to a sign to be used for the edge. This parameter is only used with the 'signed' option. See IndraNet.to_signed_graph for more info.

- **belief_flattening** (*str or function(networkx.DiGraph, edge)*) – The method to use when updating the belief for the flattened edge.

  If a string is provided, it must be one of the predefined options 'simple_scorer' or 'complementary_belief'.

  If a function is provided, it must take the flattened graph 'G' and an edge 'edge' to perform the belief flattening on and return a number:

```
>>> def belief_flattening(G, edge):
...     # Return the average belief score of the constituent␣
↪edges
...     all_beliefs = [s['belief']
...         for s in G.edges[edge]['statements']]
...     return sum(all_beliefs)/len(all_beliefs)
```

- **weight_flattening** (*function(networkx.DiGraph)*) – A function taking at least the graph G as an argument and returning G after adding edge weights as an edge attribute to the flattened edges using the reserved keyword 'weight'.

  Example:

```
>>> def weight_flattening(G):
...     # Sets the flattened weight to the average of the
...     # inverse source count
...     for edge in G.edges:
...         w = [1/s['evidence_count']
...             for s in G.edges[edge]['statements']]
...         G.edges[edge]['weight'] = sum(w)/len(w)
...     return G
```

- **keep_self_loops** (*Optional[bool]*) – Whether to keep the self-loops when constructing the graph.

Returns **model** – IndraNet graph object.

Return type *IndraNet*

indra.assemblers.indranet.assembler.**get_ag_ns_id**(*ag*)
Return a tuple of name space, id from an Agent's db_refs.

4.10 Explanation (indra.explanation)

4.10.1 Check whether a model satisfies a property (indra.explanation.model_checker)

Shared Model Checking Functionality (indra.explanation.model_checker.model_checker)

class indra.explanation.model_checker.model_checker.**ModelChecker**(*model*, *statements=None*, *do_sampling=False*, *seed=None*, *nodes_to_agents=None*)

The parent class of all ModelCheckers.

Parameters

- **model** (*pysb.Model or indra.assemblers.indranet.IndraNet or PyBEL.Model*) – Depending on the ModelChecker class, can be different type.
- **statements** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to check the model against.
- **do_sampling** (*bool*) – Whether to use breadth-first search or weighted sampling to generate paths. Default is False (breadth-first search).
- **seed** (*int*) – Random seed for sampling (optional, default is None).
- **nodes_to_agents** (*dict*) – A dictionary mapping nodes of intermediate signed edges graph to INDRA agents.

graph
A DiGraph with signed nodes to find paths in.

Type nx.Digraph

add_statements(*stmts*)
Add to the list of statements to check against the model.

Parameters **stmts** (*list[indra.statements.Statement]*) – The list of Statements to be added for checking.

check_model(*max_paths=1*, *max_path_length=5*, *agent_filter_func=None*, *edge_filter_func=None*, *allow_direct=True*)
Check all the statements added to the ModelChecker.

Parameters

- **max_paths** (*Optional[int]*) – The maximum number of specific paths to return for each Statement to be explained. Default: 1
- **max_path_length** (*Optional[int]*) – The maximum length of specific paths to return. Default: 5
- **agent_filter_func** (*Optional[function]*) – A function to constrain the intermediate nodes in the path. A function should take an agent as a parameter and return True if the agent is allowed to be in a path and False otherwise.
- **edge_filter_func** (*Optional[function]*) – A function to filter out edges from the graph. A function should take nodes (and key in case of MultiGraph) as parameters and return True if an edge can be in the graph and False if it should be filtered out.
- **allow_direct** (*Optional[bool]*) – Whether to allow direct path of length 1 (edge between source and target) to be returned as a result. Default: True.

Returns Each tuple contains the Statement checked against the model and a PathResult object describing the results of model checking.

Return type list of (*Statement*, *PathResult*)

check_statement(*stmt*, *max_paths=1*, *max_path_length=5*, *agent_filter_func=None*, *node_filter_func=None*, *edge_filter_func=None*, *allow_direct=True*)
Check a single Statement against the model.

Parameters

- **stmt** (*indra.statements.Statement*) – The Statement to check.
- **max_paths** (*Optional[int]*) – The maximum number of specific paths to return for each Statement to be explained. Default: 1

- **max_path_length** (*Optional[int]*) – The maximum length of specific paths to return. Default: 5
- **agent_filter_func** (*Optional[function]*) – A function to constrain the intermediate nodes in the path. A function should take an agent as a parameter and return True if the agent is allowed to be in a path and False otherwise.
- **node_filter_func** (*Optional[function]*) – Similar to agent_filter_func but it takes a node as a parameter instead of agent. If not provided, node_filter_func will be generated from agent_filter_func.
- **edge_filter_func** (*Optional[function]*) – A function to filter out edges from the graph. A function should take nodes (and key in case of MultiGraph) as parameters and return True if an edge can be in the graph and False if it should be filtered out.
- **allow_direct** (*Optional[bool]*) – Whether to allow direct path of length 1 (edge between source and target) to be returned as a result. Default: True.

Returns **result** – A PathResult object containing the result of a test.

Return type indra.explanation.modelchecker.PathResult

find_paths(*subj*, *obj*, *max_paths=1*, *max_path_length=5*, *loop=False*, *filter_func=None*, *allow_direct=True*)

Check for a source/target path in the model.

Parameters

- **subj** (*indra.explanation.model_checker.NodesContainer*) – NodesContainer representing test statement subject.
- **obj** (*indra.explanation.model_checker.NodesContainer*) – NodesContainer representing test statement object.
- **max_paths** (*int*) – The maximum number of specific paths to return.
- **max_path_length** (*int*) – The maximum length of specific paths to return.
- **loop** (*bool*) – Whether we are looking for a loop path.
- **filter_func** (*function or None*) – A function to constrain the search. A function should take a node as a parameter and return True if the node is allowed to be in a path and False otherwise. If None, then no filtering is done.
- **allow_direct** (*Optional[bool]*) – Whether to allow direct path of length 1 (edge between source and target) to be returned as a result. Default: True.

Returns PathResult object indicating the results of the attempt to find a path.

Return type *PathResult*

get_graph(***kwargs*)

Return a graph with signed nodes to find the path.

get_nodes_to_agents(**args*, ***kwargs*)

Return a dictionary mapping nodes of intermediate signed edges graph to INDRA agents.

get_ref(*ag*, *node*, *rel*)

Create a refinement edge.

process_statement(*stmt*)

This method processes the test statement to get the data about subject and object, according to the specific model requirements for model checking, e.g. PysbModelChecker gets subject monomer patterns and ob-

servables, while graph based ModelCheckers will return signed nodes corresponding to subject and object. If any of the requirements are not satisfied, result code is also returned to construct PathResult object.

Parameters **stmt** (*indra.statements.Statement*) – A statement to process.

Returns

- **subj_data** (*NodesContainer*) – NodesContainer for statement subject.
- **obj_data** (*NodesContainer*) – NodesContainer for statement object.
- **result_code** (*str or None*) – Result code to construct PathResult.

process_subject(*subj_data*)

Processes the subject of the test statement and returns the necessary information to check the statement. In case of PysbModelChecker, method returns input_rule_set. If any of the requirements are not satisfied, result code is also returned to construct PathResult object.

update_filter_func(*agent_filter_func*)

Converts a function filtering agents to a function filtering nodes

Parameters **agent_filter_func** (*function*) – A function to constrain the intermediate nodes in the path. A function should take an agent as a parameter and return True if the agent is allowed to be in a path and False otherwise.

Returns **node_filter_func** – A new filter function applying the logic from agent_filter_func to nodes instead of agents.

Return type function

class indra.explanation.model_checker.model_checker.**NodesContainer**(*main_agent*, *ref_agents=None*)

Contains the information about nodes corresponding to a given agent of the test statement.

Parameters

- **main_agent** (*indra.statements.Agent*) – An INDRA agent representing a subject or object of test statement.
- **ref_agents** (*list[indra.statements.Agent]*) – A list of agents that are refinements of main agent.

main_nodes

A list of nodes corresponding to main agent.

Type list[tuple]

ref_nodes

A list of nodes corresponding to refinement agents.

Type list[tuple]

all_nodes

A list of all nodes corresponding to main agent or its refinements.

Type list[tuple]

common_target

Common target node connected to all nodes. If there's only one node in all_nodes, then common_target is not used.

Type tuple or None

main_interm
A list of intermediate representation between main agent and main nodes (only used in PySB currently - MonomerPatterns).

Type list[MonomerPattern]

ref_interm
A list of intermediate representation between ref_agents and ref_nodes (only used in PySB currently - MonomerPatterns).

Type list[MonomerPattern]

get_all_nodes()
Combine main and refinement nodes for pathfinding.

get_total_nodes()
Get total number of nodes in this container.

is_ref(*node*)
Whether a given node is a refinement node.

class indra.explanation.model_checker.model_checker.**PathMetric**(*source_node*, *target_node*, *length*)
Describes results of simple path search (path existence).

source_node
The source node of the path

Type str

target_node
The target node of the path

Type str

length
The length of the path

Type int

class indra.explanation.model_checker.model_checker.**PathResult**(*path_found*, *result_code*, *max_paths*, *max_path_length*)
Describes results of running the ModelChecker on a single Statement.

path_found
True if a path was found, False otherwise.

Type bool

result_code

- *STATEMENT_TYPE_NOT_HANDLED* - The provided statement type is not handled
- *SUBJECT_MONOMERS_NOT_FOUND* or *SUBJECT_NOT_FOUND* - Statement subject not found in model
- *OBSERVABLES_NOT_FOUND* or *OBJECT_NOT_FOUND* - Statement has no associated observable
- *NO_PATHS_FOUND* - Statement has no path for any observable
- *MAX_PATH_LENGTH_EXCEEDED* - Statement has no path len <= MAX_PATH_LENGTH
- *PATHS_FOUND* - Statement has path len <= MAX_PATH_LENGTH
- *INPUT_RULES_NOT_FOUND* - No rules with Statement subject found

- *MAX_PATHS_ZERO* - Path found but MAX_PATHS is set to zero

Type string

max_paths
The maximum number of specific paths to return for each Statement to be explained.

Type int

max_path_length
The maximum length of specific paths to return.

Type int

path_metrics
A list of PathMetric objects, each describing the results of a simple path search (path existence).

Type list[indra.explanation.model_checker.PathMetric]

paths
A list of paths obtained from path finding. Each path is a list of tuples (which are edges in the path), with the first element of the tuple the name of a rule, and the second element its polarity in the path.

Type list[list[tuple[str, int]]]

indra.explanation.model_checker.model_checker.**prune_signed_nodes**(*graph*)
Prune nodes with sign (1) if they do not have predecessors.

indra.explanation.model_checker.model_checker.**signed_edges_to_signed_nodes**(*graph*, *prune_nodes=True*, *edge_signs={'neg': 1, 'pos': 0}*, *copy_edge_data=False*)
Convert a graph with signed edges to a graph with signed nodes.

Each pair of nodes linked by an edge in an input graph are represented as four nodes and two edges in the new graph. For example, an edge (a, b, 0), where a and b are nodes and 0 is a sign of an edge (positive), will be represented as edges ((a, 0), (b, 0)) and ((a, 1), (b, 1)), where (a, 0), (a, 1), (b, 0), (b, 1) are signed nodes. An edge (a, b, 1) with sign 1 (negative) will be represented as edges ((a, 0), (b, 1)) and ((a, 1), (b, 0)).

Parameters

- **graph** (*networkx.MultiDiGraph*) – Graph with signed edges to convert. Can have multiple edges between a pair of nodes.
- **prune_nodes** (*Optional[bool]*) – If True, iteratively prunes negative (with sign 1) nodes without predecessors.
- **edge_signs** (*dict*) – A dictionary representing the signing policy of incoming graph. The dictionary should have strings 'pos' and 'neg' as keys and integers as values.
- **copy_edge_data** (*bool|set(keys)*) – Option for copying edge data as well from graph. If False (default), no edge data is copied (except sign). If True, all edge data is copied. If a set of keys is provided, only the keys appearing in the set will be copied, assuming the key is part of a nested dictionary.

Returns **signed_nodes_graph**

Return type networkx.DiGraph

Checking PySB model (indra.explanation.model_checker.pysb)

class indra.explanation.model_checker.pysb.**PysbModelChecker**(*model*, *statements=None*, *agent_obs=None*, *do_sampling=False*, *seed=None*, *model_stmts=None*, *nodes_to_agents=None*)

Check a PySB model against a set of INDRA statements.

Parameters

- **model** (*pysb.Model*) – A PySB model to check.
- **statements** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to check the model against.
- **agent_obs** (*Optional[list[indra.statements.Agent]]*) – A list of INDRA Agents in a given state to be observed.
- **do_sampling** (*bool*) – Whether to use breadth-first search or weighted sampling to generate paths. Default is False (breadth-first search).
- **seed** (*int*) – Random seed for sampling (optional, default is None).
- **model_stmts** (*list[indra.statements.Statement]*) – A list of INDRA statements used to assemble PySB model.
- **nodes_to_agents** (*dict*) – A dictionary mapping nodes of intermediate signed edges graph to INDRA agents.

graph

A DiGraph with signed nodes to find paths in.

Type nx.Digraph

draw_im(*fname*)

Draw and save the influence map in a file.

Parameters **fname** (*str*) – The name of the file to save the influence map in. The extension of the file will determine the file format, typically png or pdf.

generate_im(*model*)

Return a graph representing the influence map generated by Kappa

Parameters **model** (*pysb.Model*) – The PySB model whose influence map is to be generated

Returns **graph** – A MultiDiGraph representing the influence map

Return type networkx.MultiDiGraph

get_all_mps(*agents*, *ignore_activities=False*, *mapping=False*)

Get a list of all monomer patterns for a list of agents.

get_graph(*prune_im=True*, *prune_im_degrade=True*, *prune_im_subj_obj=False*, *add_namespaces=False*, *edge_filter_func=None*)

Get influence map and convert it to a graph with signed nodes.

get_im(*force_update=False*)

Get the influence map for the model, generating it if necessary.

Parameters **force_update** (*bool*) – Whether to generate the influence map when the function is called. If False, returns the previously generated influence map if available. Defaults to True.

Returns

The influence map can be rendered as a pdf using the dot layout program as follows:

```
im_agraph = nx.nx_agraph.to_agraph(influence_map)
im_agraph.draw('influence_map.pdf', prog='dot')
```

Return type networkx MultiDiGraph object containing the influence map.

`get_nodes_to_agents`(*add_namespaces=False*)

Return a dictionary mapping influence map nodes to INDRA agents.

Parameters **add_namespaces** (*bool*) – Whether to propagate namespaces to node data. Default: False.

Returns **nodes_to_agents** – A dictionary mapping influence map nodes to INDRA agents.

Return type dict

`get_refinements`(*agent*)

Return a list of refinement agents that are part of the model.

`process_statement`(*stmt*)

This method processes the test statement to get the data about subject and object, according to the specific model requirements for model checking, e.g. PysbModelChecker gets subject monomer patterns and observables, while graph based ModelCheckers will return signed nodes corresponding to subject and object. If any of the requirements are not satisfied, result code is also returned to construct PathResult object.

Parameters **stmt** (*indra.statements.Statement*) – A statement to process.

Returns

- **subj_data** (*NodesContainer*) – NodesContainer for statement subject.
- **obj_data** (*NodesContainer*) – NodesContainer for statement object.
- **result_code** (*str or None*) – Result code to construct PathResult.

`process_subject`(*subj_mp*)

Processes the subject of the test statement and returns the necessary information to check the statement. In case of PysbModelChecker, method returns input_rule_set. If any of the requirements are not satisfied, result code is also returned to construct PathResult object.

`prune_influence_map`()

Remove edges between rules causing problematic non-transitivity.

First, all self-loops are removed. After this initial step, edges are removed between rules when they share *all* child nodes except for each other; that is, they have a mutual relationship with each other and share all of the same children.

Note that edges must be removed in batch at the end to prevent edge removal from affecting the lists of rule children during the comparison process.

`prune_influence_map_degrade_bind_positive`(*model_stmts*)

Prune positive edges between X degrading and X forming a complex with Y.

`prune_influence_map_subj_obj`()

Prune influence map to include only edges where the object of the upstream rule matches the subject of the downstream rule.

`score_paths`(*paths*, *agents_values*, *loss_of_function=False*, *sigma=0.15*, *include_final_node=False*)

Return scores associated with a given set of paths.

Parameters

- **paths** (*list[list[tuple[str, int]]]*) – A list of paths obtained from path finding. Each path is a list of tuples (which are edges in the path), with the first element of the tuple the name of a rule, and the second element its polarity in the path.
- **agents_values** (*dict[indra.statements.Agent, float]*) – A dictionary of INDRA Agents and their corresponding measured value in a given experimental condition.
- **loss_of_function** (*Optional[boolean]*) – If True, flip the polarity of the path. For instance, if the effect of an inhibitory drug is explained, set this to True. Default: False
- **sigma** (*Optional[float]*) – The estimated standard deviation for the normally distributed measurement error in the observation model used to score paths with respect to data. Default: 0.15
- **include_final_node** (*Optional[boolean]*) – Determines whether the final node of the path is included in the score. Default: False

indra.explanation.model_checker.pysb.**remove_im_params**(*model*, *im*)

Remove parameter nodes from the influence map.

Parameters

- **model** (*pysb.core.Model*) – PySB model.
- **im** (*networkx.MultiDiGraph*) – Influence map.

Returns Influence map with the parameter nodes removed.

Return type networkx.MultiDiGraph

Checking Signed Graph (indra.explanation.model_checker.signed_graph)

class indra.explanation.model_checker.signed_graph.**SignedGraphModelChecker**(*model*, *statements=None*, *do_sampling=False*, *seed=None*, *nodes_to_agents=None*)

Check an signed MultiDiGraph against a set of INDRA statements.

Parameters

- **model** (*networkx.MultiDiGraph*) – Signed MultiDiGraph to check.
- **statements** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to check the model against.
- **do_sampling** (*bool*) – Whether to use breadth-first search or weighted sampling to generate paths. Default is False (breadth-first search).
- **seed** (*int*) – Random seed for sampling (optional, default is None).
- **nodes_to_agents** (*dict*) – A dictionary mapping nodes of intermediate signed edges graph to INDRA agents.

graph

A DiGraph with signed nodes to find paths in.

Type nx.Digraph

get_graph(*edge_filter_func=None*, *copy_edge_data=None*)

Get a signed nodes graph to search for paths in.

Parameters

- **edge_filter_func** (*Optional[function]*) – A function to filter out edges from the graph. A function should take nodes (and key in case of MultiGraph) as parameters and return True if an edge can be in the graph and False if it should be filtered out.
- **copy_edge_data** (*set(str)*) – A set of keys to copy from original model edge data to the graph edge data. If None, only belief data is copied by default.

get_nodes(*agent*, *graph*, *target_polarity*)
Get all nodes corresponding to a given agent.

get_nodes_to_agents()
Return a dictionary mapping IndraNet nodes to INDRA agents.

process_statement(*stmt*)
This method processes the test statement to get the data about subject and object, according to the specific model requirements for model checking, e.g. PysbModelChecker gets subject monomer patterns and observables, while graph based ModelCheckers will return signed nodes corresponding to subject and object. If any of the requirements are not satisfied, result code is also returned to construct PathResult object.

Parameters **stmt** (*indra.statements.Statement*) – A statement to process.

Returns

- **subj_data** (*NodesContainer*) – NodesContainer for statement subject.
- **obj_data** (*NodesContainer*) – NodesContainer for statement object.
- **result_code** (*str or None*) – Result code to construct PathResult.

Checking Unsigned Graph (indra.explanation.model_checker.unsigned_graph)

class indra.explanation.model_checker.unsigned_graph.**UnsignedGraphModelChecker**(*model*, *statements=None*, *do_sampling=False*, *seed=None*, *nodes_to_agents=None*)

Check an unsigned DiGraph against a set of INDRA statements.

Parameters

- **model** (*networkx.DiGraph*) – Unsigned DiGraph to check.
- **statements** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to check the model against.
- **do_sampling** (*bool*) – Whether to use breadth-first search or weighted sampling to generate paths. Default is False (breadth-first search).
- **seed** (*int*) – Random seed for sampling (optional, default is None).
- **nodes_to_agents** (*dict*) – A dictionary mapping nodes of intermediate signed edges graph to INDRA agents.

graph
A DiGraph with signed nodes to find paths in.

Type nx.Digraph

get_graph(*edge_filter_func=None*, *copy_edge_data=None*)
Get a signed nodes graph to search for paths in.

Parameters

- **edge_filter_func** (*Optional[function]*) – A function to filter out edges from the graph. A function should take nodes (and key in case of MultiGraph) as parameters and return True if an edge can be in the graph and False if it should be filtered out.
- **copy_edge_data** (*set(str)*) – A set of keys to copy from original model edge data to the graph edge data. If None, only belief data is copied by default.

get_nodes(*agent*, *graph*)
Get all nodes corresponding to a given agent.

get_nodes_to_agents()
Return a dictionary mapping IndraNet nodes to INDRA agents.

process_statement(*stmt*)
This method processes the test statement to get the data about subject and object, according to the specific model requirements for model checking, e.g. PysbModelChecker gets subject monomer patterns and observables, while graph based ModelCheckers will return signed nodes corresponding to subject and object. If any of the requirements are not satisfied, result code is also returned to construct PathResult object.

Parameters **stmt** (*indra.statements.Statement*) – A statement to process.

Returns

- **subj_data** (*NodesContainer*) – NodesContainer for statement subject.
- **obj_data** (*NodesContainer*) – NodesContainer for statement object.
- **result_code** (*str or None*) – Result code to construct PathResult.

Checking PyBEL Graph (indra.explanation.model_checker.pybel)

class indra.explanation.model_checker.pybel.**PybelModelChecker**(*model*, *statements=None*, *do_sampling=False*, *seed=None*, *nodes_to_agents=None*)
Check a PyBEL model against a set of INDRA statements.

Parameters

- **model** (*pybel.BELGraph*) – A Pybel model to check.
- **statements** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to check the model against.
- **do_sampling** (*bool*) – Whether to use breadth-first search or weighted sampling to generate paths. Default is False (breadth-first search).
- **seed** (*int*) – Random seed for sampling (optional, default is None).
- **nodes_to_agents** (*dict*) – A dictionary mapping nodes of intermediate signed edges graph to INDRA agents.

graph
A DiGraph with signed nodes to find paths in.

Type nx.Digraph

get_graph(*include_variants=False*, *symmetric_variant_links=False*, *include_components=True*, *symmetric_component_links=True*, *edge_filter_func=None*)
Convert a PyBELGraph to a graph with signed nodes.

get_nodes(*agent*, *graph*, *target_polarity*)
Get all nodes corresponding to a given agent.

get_nodes_to_agents()
Return a dictionary mapping PyBEL nodes to INDRA agents.

process_statement(*stmt*)
This method processes the test statement to get the data about subject and object, according to the specific model requirements for model checking, e.g. PysbModelChecker gets subject monomer patterns and observables, while graph based ModelCheckers will return signed nodes corresponding to subject and object. If any of the requirements are not satisfied, result code is also returned to construct PathResult object.

Parameters **stmt** (*indra.statements.Statement*) – A statement to process.

Returns

- **subj_data** (*NodesContainer*) – NodesContainer for statement subject.
- **obj_data** (*NodesContainer*) – NodesContainer for statement object.
- **result_code** (*str or None*) – Result code to construct PathResult.

4.10.2 Path finding algorithms for explanation (indra.explanation.pathfinding)

Path finding functions (indra.explanation.pathfinding.pathfinding)

indra.explanation.pathfinding.pathfinding.**bfs_search**(*g*, *source_node*, *reverse=False*, *depth_limit=2*, *path_limit=None*, *max_per_node=5*, *node_filter=None*, *node_blacklist=None*, *terminal_ns=None*, *sign=None*, *max_memory=536870912*, *hashes=None*, *allow_edge=None*, *strict_mesh_id_filtering=False*, *edge_filter=None*, ***kwargs*)

Do breadth first search from a given node and yield paths

Parameters

- **g** (DiGraph) – An nx.DiGraph to search in. Can also be a signed node graph. It is required that node data contains 'ns' (namespace) and edge data contains 'belief'.
- **source_node** (Union[str, Tuple[str, int]]) – Node in the graph to start from.
- **reverse** (Optional[bool]) – If True go upstream from source, otherwise go downstream. Default: False.
- **depth_limit** (Optional[int]) – Stop when all paths with this many edges have been found. Default: 2.
- **path_limit** (Optional[int]) – The maximum number of paths to return. Default: no limit.
- **max_per_node** (Optional[int]) – The maximum number of paths to yield per parent node. If 1 is chosen, the search only goes down to the leaf node of its first encountered branch. Default: 5
- **node_filter** (Optional[List[str]]) – The allowed namespaces (node attribute 'ns') for the nodes in the path
- **node_blacklist** (Optional[Set[Union[str, Tuple[str, int]]]]) – A set of nodes to ignore. Default: None.

- **terminal_ns** (Optional[List[str]]) – Force a path to terminate when any of the namespaces in this list are encountered and only yield paths that terminate at these namespaces
- **sign** (Optional[int]) – If set, defines the search to be a signed search. Default: None.
- **max_memory** (Optional[int]) – The maximum memory usage in bytes allowed for the variables queue and visited. Default: 1073741824 bytes (== 1 GiB).
- **hashes** (Optional[List[int]]) – List of hashes used (if not empty) to select edges for path finding
- **allow_edge** (Optional[Callable[[Union[str, Tuple[str, int]], Union[str, Tuple[str, int]]], bool]]) – Function telling the edge must be omitted
- **strict_mesh_id_filtering** (Optional[bool]) – If true, exclude all edges not relevant to provided hashes
- **edge_filter** (Optional[Callable[[DiGraph, Union[str, Tuple[str, int]], Union[str, Tuple[str, int]]], bool]]) – If provided, must be a function that takes three arguments: a graph g, and the nodes u, v of the edge between u and v. The function must return a boolean. The function must return True if the edge is allowed, otherwise False. Example of function that only allows edges that have an edge belief above 0.75:

```
>>> g = nx.DiGraph({'CHEK1': {'FANC': {'belief': 1}}})
>>> def filter_example(g, u, v):
...     return g.edges[u, v].get('belief', 0) > 0.75
>>> path_generator = bfs_search(g, source_node='CHEK1',
...                             edge_filter=filter_example)
```

Yields *Tuple[Node, ...]* – Paths in the bfs search starting from *source*.

Raises StopIteration – Raises StopIteration when no more paths are available or when the memory limit is reached

Return type Generator[Tuple[Union[str, Tuple[str, int]]], Tuple[Optional[Set[Union[str, Tuple[str, int]]]], Optional[Set[Tuple[Union[str, Tuple[str, int]], Union[str, Tuple[str, int]]]]], None]

indra.explanation.pathfinding.pathfinding.**bfs_search_multiple_nodes**(*g*, *source_nodes*, *path_limit=None*, ***kwargs*)

Do breadth first search from each of given nodes and yield paths until path limit is met.

Parameters

- **g** (*nx.Digraph*) – An nx.DiGraph to search in. Can also be a signed node graph. It is required that node data contains 'ns' (namespace) and edge data contains 'belief'.
- **source_nodes** (*list[node]*) – List of nodes in the graph to start from.
- **path_limit** (*int*) – The maximum number of paths to return. Default: no limit.
- ****kwargs** (*keyword arguments*) – Any kwargs to pass to bfs_search.

Yields **path** (*tuple(node)*) – Paths in the bfs search starting from *source*.

indra.explanation.pathfinding.pathfinding.**find_sources**(*graph*, *target*, *sources*, *filter_func=None*)

Get the set of source nodes with paths to the target.

Given a common target and a list of sources (or None if test statement subject is None), perform a breadth-first search upstream from the target to determine whether there are any sources that have paths to the target. For efficiency, does not return the full path, but identifies the upstream sources and the length of the path.

Parameters

- **graph** (*nx.DiGraph*) – A DiGraph with signed nodes to find paths in.
- **target** (*node*) – The signed node (usually common target node) in the graph to start looking upstream for matching sources.
- **sources** (*list[node]*) – Signed nodes corresponding to the subject or upstream influence being checked.
- **filter_func** (*Optional[function]*) – A function to constrain the intermediate nodes in the path. A function should take a node as a parameter and return True if the node is allowed to be in a path and False otherwise.

Returns Yields tuples of source node and path length (int). If there are no paths to any of the given source nodes, the generator is empty.

Return type generator of (source, path_length)

indra.explanation.pathfinding.pathfinding.**get_path_iter**(*graph*, *source*, *target*, *path_length*, *loop*, *dummy_target*, *filter_func*)

Return a generator of paths with path_length cutoff from source to target.

Parameters

- **graph** (*nx.Digraph*) – An nx.DiGraph to search in.
- **source** (*node*) – Starting node for path.
- **target** (*node*) – Ending node for path.
- **path_length** (*int*) – Maximum depth of the paths.
- **loop** (*bool*) – Whether the path should be a loop. If True, source is appended to path.
- **dummy_target** (*bool*) – Whether provided target is a dummy node and should be removed from path
- **filter_func** (*function or None*) – A function to constrain the search. A function should take a node as a parameter and return True if the node is allowed to be in a path and False otherwise. If None, then no filtering is done.

Returns **path_generator** – A generator of the paths between source and target.

Return type generator

indra.explanation.pathfinding.pathfinding.**open_dijkstra_search**(*g*, *start*, *reverse=False*, *path_limit=None*, *node_filter=None*, *hashes=None*, *ignore_nodes=None*, *ignore_edges=None*, *terminal_ns=None*, *weight=None*, *ref_counts_function=None*, *const_c=1*, *const_tk=10*)

Do Dijkstra search from a given node and yield paths

Parameters

- **g** (*nx.Digraph*) – An nx.DiGraph to search in.
- **start** (*node*) – Node in the graph to start from.
- **reverse** (*bool*) – If True go upstream from source, otherwise go downstream. Default: False.

- **path_limit** (*int*) – The maximum number of paths to return. Default: no limit.
- **node_filter** (*list[str]*) – The allowed namespaces (node attribute 'ns') for the nodes in the path
- **hashes** (*list*) – List of hashes used to set edge weights
- **ignore_nodes** (*container of nodes*) – nodes to ignore, optional
- **ignore_edges** (*container of edges*) – edges to ignore, optional
- **terminal_ns** (*list[str]*) – Force a path to terminate when any of the namespaces in this list are encountered and only yield paths that terminate at these namepsaces
- **weight** (*str*) – Name of edge's attribute used as its weight
- **ref_counts_function** (*function*) – function counting references and PMIDs of an edge from its statement hashes
- **const_c** (*int*) – Constant used in MeSH IDs-based weight calculation
- **const_tk** (*int*) – Constant used in MeSH IDs-based weight calculation

Yields **path** (*tuple(node)*) – Paths in the bfs search starting from *source*.

indra.explanation.pathfinding.pathfinding.**shortest_simple_paths**(*G*, *source*, *target*, *weight=None*, *ignore_nodes=None*, *ignore_edges=None*, *hashes=None*, *ref_counts_function=None*, *strict_mesh_id_filtering=False*, *const_c=1*, *const_tk=10*)

Generate all simple paths in the graph G from source to target, starting from shortest ones.

A simple path is a path with no repeated nodes.

If a weighted shortest path search is to be used, no negative weights are allowed.

Parameters

- **G** (*NetworkX graph*) –
- **source** (*node*) – Starting node for path
- **target** (*node*) – Ending node for path
- **weight** (*string*) – Name of the edge attribute to be used as a weight. If None all edges are considered to have unit weight. Default value None.
- **ignore_nodes** (*container of nodes*) – nodes to ignore, optional
- **ignore_edges** (*container of edges*) – edges to ignore, optional
- **hashes** (*list*) – hashes specifying (if not empty) allowed edges
- **ref_counts_function** (*function*) – function counting references and PMIDs of an edge from its statement hashes
- **strict_mesh_id_filtering** (*bool*) – if true, exclude all edges not relevant to provided hashes
- **const_c** (*int*) – Constant used in MeSH IDs-based weight calculation
- **const_tk** (*int*) – Constant used in MeSH IDs-based weight calculation

Returns **path_generator** – A generator that produces lists of simple paths, in order from shortest to longest.

Return type generator

Raises

- **NetworkXNoPath** – If no path exists between source and target.
- **NetworkXError** – If source or target nodes are not in the input graph.
- **NetworkXNotImplemented** – If the input graph is a Multi[Di]Graph.

Examples

```
>>> G = nx.cycle_graph(7)
>>> paths = list(nx.shortest_simple_paths(G, 0, 3))
>>> print(paths)
[[0, 1, 2, 3], [0, 6, 5, 4, 3]]
```

You can use this function to efficiently compute the k shortest/best paths between two nodes.

```
>>> from itertools import islice
>>> def k_shortest_paths(G, source, target, k, weight=None):
...     return list(islice(nx.shortest_simple_paths(G, source, target,
...         weight=weight), k))
>>> for path in k_shortest_paths(G, 0, 3, 2):
...     print(path)
[0, 1, 2, 3]
[0, 6, 5, 4, 3]
```

Notes

This procedure is based on algorithm by Jin Y. Yen[1]. Finding the first K paths requires $O(KN^3)$ operations.

See also:

`all_shortest_paths`, `shortest_path`, `all_simple_paths`

References

Path finding utilities (`indra.explanation.pathfinding.util`)

`indra.explanation.pathfinding.util.`**`get_sorted_neighbors`**(*G*, *node*, *reverse*, *force_edges=None*, *edge_filter=None*)

Filter and sort neighbors of a node in descending order by belief

Parameters

- **G** (`DiGraph`) – A networkx DiGraph
- **node** (`Union[str, Tuple[str, int]]`) – A valid node name or signed node name
- **reverse** (`bool`) – Indicates direction of search. Neighbors are either successors (downstream search) or predecessors (reverse search).

[1] Jin Y. Yen, "Finding the K Shortest Loopless Paths in a Network", Management Science, Vol. 17, No. 11, Theory Series (Jul., 1971), pp. 712-716.

- **force_edges** (Optional[List[Tuple[Union[str, Tuple[str, int]], Union[str, Tuple[str, int]]]]]) – A list of allowed edges. If provided, only allow neighbors that can be reached by the allowed edges.
- **edge_filter** (Optional[Callable[[DiGraph, Union[str, Tuple[str, int]], Union[str, Tuple[str, int]]], bool]]) – If provided, must be a function that takes three arguments: a graph g, and the nodes u, v of the edge between u and v. The function must return a boolean. The function must return True if the edge is allowed, otherwise False.

Returns A list of nodes representing the filtered and sorted neighbors

Return type List[Node]

indra.explanation.pathfinding.util.**get_subgraph**(*g*, *edge_filter_func*)
Get a subgraph of original graph filtered by a provided function.

indra.explanation.pathfinding.util.**path_sign_to_signed_nodes**(*source*, *target*, *edge_sign*)
Translates a signed edge or path to valid signed nodes

Pairs with a negative source node are filtered out.

Parameters

- **source** (*str|int*) – The source node
- **target** (*str|int*) – The target node
- **edge_sign** (*int*) – The sign of the edge

Returns **sign_tuple** – Tuple of tuples of the valid combination of signed nodes

Return type (a, sign), (b, sign)

indra.explanation.pathfinding.util.**signed_nodes_to_signed_edge**(*source*, *target*)
Create the triple (node, node, sign) from a pair of signed nodes

Assuming source, target forms an edge of signed nodes: edge = (a, sign), (b, sign), return the corresponding signed edge triple

Parameters

- **source** (*tuple(str|int, sign)*) – A valid signed node
- **target** (*tuple(str|int, sign)*) – A valid signed node

Returns A tuple, (source, target, sign), representing the corresponding signed edge.

Return type tuple

4.10.3 Reporting explanations (indra.explanation.reporting)

class indra.explanation.reporting.**PybelEdge**(*source*, *target*, *relation*, *reverse*)

property relation
Alias for field number 2

property reverse
Alias for field number 3

property source
Alias for field number 0

property target
Alias for field number 1

class indra.explanation.reporting.**RefEdge**(*source*, *relation*, *target*)
Refinement edge representing ontological relationship between nodes.

Parameters

- **source** (*indra.statements.Agent*) – Source agent of the edge.
- **target** (*indra.statements.Agent*) – Target agent of the edge.
- **relation** (*str*) – 'is_ref' or 'has_ref' depending on the direction.

indra.explanation.reporting.**stmt_from_rule**(*rule_name*, *model*, *stmts*)
Return the source INDRA Statement corresponding to a rule in a model.

Parameters

- **rule_name** (*str*) – The name of a rule in the given PySB model.
- **model** (*pysb.core.Model*) – A PySB model which contains the given rule.
- **stmts** (*list[indra.statements.Statement]*) – A list of INDRA Statements from which the model was assembled.

Returns **stmt** – The Statement from which the given rule in the model was obtained.

Return type indra.statements.Statement

indra.explanation.reporting.**stmts_from_indranet_path**(*path*, *model*, *signed*, *from_db=True*, *stmts=None*)
Return source Statements corresponding to a path in an IndraNet model (found by SignedGraphModelChecker or UnsignedGraphModelChecker).

Parameters

- **path** (*list[tuple[str, int]]*) – A list of tuples where the first element of the tuple is the name of an agent, and the second is the associated polarity along a path.
- **model** (*nx.Digraph or nx.MultiDiGraph*) – An IndraNet model flattened into an unsigned DiGraph or signed MultiDiGraph.
- **signed** (*bool*) – Whether the model and path are signed.
- **from_db** (*bool*) – If True, uses statement hashes to query the database. Otherwise, looks for path statements in provided stmts.
- **stmts** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements from which the model was assembled. Required if from_db is set to False.

Returns **path_stmts** – A list of lists of INDRA statements explaining the path (each inner corresponds to one step in the path because the flattened model can have multiple statements per edge).

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

indra.explanation.reporting.**stmts_from_pybel_path**(*path*, *model*, *from_db=True*, *stmts=None*)
Return source Statements corresponding to a path in a PyBEL model.

Parameters

- **path** (*list[tuple[str, int]]*) – A list of tuples where the first element of the tuple is the name of an agent, and the second is the associated polarity along a path.
- **model** (*pybel.BELGraph*) – A PyBEL BELGraph model.
- **from_db** (*bool*) – If True, uses statement hashes to query the database. Otherwise, looks for path statements in provided stmts.

- **stmts** (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements from which the model was assembled. Required if from_db is set to False.

Returns **path_stmts** – A list of lists of INDRA statements explaining the path (each inner corresponds to one step in the path because PyBEL model can have multiple edges representing multiple statements and evidences between two nodes).

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

indra.explanation.reporting.**stmts_from_pysb_path**(*path*, *model*, *stmts*)

Return source Statements corresponding to a path in a model.

Parameters

- **path** (*list[tuple[str, int]]*) – A list of tuples where the first element of the tuple is the name of a rule, and the second is the associated polarity along a path.
- **model** (*pysb.core.Model*) – A PySB model which contains the rules along the path.
- **stmts** (*list[indra.statements.Statement]*) – A list of INDRA Statements from which the model was assembled.

Returns **path_stmts** – The Statements from which the rules along the path were obtained.

Return type list[indra.statements.Statement]

4.11 Assembly Pipeline (indra.pipeline)

class indra.pipeline.pipeline.**AssemblyPipeline**(*steps=None*)

Bases: object

An assembly pipeline that runs the specified steps on a given set of statements.

Ways to initialize and run the pipeline (examples assume you have a list of INDRA Statements stored in the *stmts* variable.)

```
>>> from indra.statements import *
>>> map2k1 = Agent('MAP2K1', db_refs={'HGNC': '6840'})
>>> mapk1 = Agent('MAPK1', db_refs={'HGNC': '6871'})
>>> braf = Agent('BRAF')
>>> stmts = [Phosphorylation(map2k1, mapk1, 'T', '185'),
...          Phosphorylation(braf, map2k1)]
```

1) Provide a JSON file containing the steps, then use the classmethod *from_json_file*, and run it with the *run* method on a list of statements. This option allows storing pipeline versions in a separate file and reproducing the same results. All functions referenced in the JSON file have to be registered with the @register_pipeline decorator.

```
>>> import os
>>> path_this = os.path.dirname(os.path.abspath(__file__))
>>> filename = os.path.abspath(
... os.path.join(path_this, '..', 'tests', 'pipeline_test.json'))
>>> ap = AssemblyPipeline.from_json_file(filename)
>>> assembled_stmts = ap.run(stmts)
```

2) Initialize a pipeline with a list of steps and run it with the *run* method on a list of statements. All functions referenced in steps have to be registered with the @register_pipeline decorator.

```
>>> steps = [
...     {"function": "filter_no_hypothesis"},
...     {"function": "filter_grounded_only",
...      "kwargs": {"score_threshold": 0.8}}
... ]
>>> ap = AssemblyPipeline(steps)
>>> assembled_stmts = ap.run(stmts)
```

3) Initialize an empty pipeline and append/insert the steps one by one. Provide a function and its args and kwargs. For arguments that require calling a different function, use the RunnableArgument class. All functions referenced here have to be either imported and passed as function objects or registered with the @register_pipeline decorator and passed as function names (strings). The pipeline built this way can be optionally saved into a JSON file. (Note that this example requires indra_world to be installed.)

```
>>> from indra.tools.assemble_corpus import *
>>> from indra_world.ontology import load_world_ontology
>>> from indra_world.belief import get_eidos_scorer
>>> ap = AssemblyPipeline()
>>> ap.append(filter_no_hypothesis)
>>> ap.append(filter_grounded_only)
>>> ap.append(run_preassembly,
...           belief_scorer=RunnableArgument(get_eidos_scorer),
...           ontology=RunnableArgument(load_world_ontology))
>>> assembled_stmts = ap.run(stmts)
>>> ap.to_json_file('filename.json')
```

Parameters **steps** (*list[dict]*) – A list of dictionaries representing steps in the pipeline. Each step should have a 'function' key and, if appropriate, 'args' and 'kwargs' keys. Arguments can be simple values (strings, integers, booleans, lists, etc.) or can be functions themselves. In case an argument is a function or a result of another function, it should also be represented as a dictionary of a similar structure. If a function itself is an argument (and not its result), the dictionary should contain a key-value pair {'no_run': True}. If an argument is a type of a statement, it should be represented as a dictionary {'stmt_type': <name of a statement type>}.

append(*func*, **args*, ***kwargs*)

Append a step to the end of the pipeline.

Args and kwargs here can be of any type. All functions referenced here have to be either imported and passed as function objects or registered with @register_pipeline decorator and passed as function names (strings). For arguments that require calling a different function, use RunnableArgument class.

Parameters

- **func** (*str or function*) – A function or the string name of a function to add to the pipeline.
- **args** (*args*) – Args that are passed to func when calling it.
- **kwargs** (*kwargs*) – Kwargs that are passed to func when calling it.

create_new_step(*func_name*, **args*, ***kwargs*)

Create a dictionary representing a new step in the pipeline.

Parameters

- **func_name** (*str*) – The string name of a function to create as a step.
- **args** (*args*) – Args that are passed to the function when calling it.

- **kwargs** (*kwargs*) – Kwargs that are passed to the function when calling it.

Returns A dict structure representing a step in the pipeline.

Return type dict

classmethod from_json_file(*filename*)
Create an instance of AssemblyPipeline from a JSON file with steps.

get_argument_value(*arg_json*)
Get a value of an argument from its json version.

static get_function_from_name(*name*)
Return a function object by name if available or raise exception.

Parameters **name** (*str*) – The name of the function.

Returns The function that was found based on its name. If not found, a NotRegisteredFunctionError is raised.

Return type function

static get_function_parameters(*func_dict*)
Retrieve a function name and arguments from function dictionary.

Parameters **func_dict** (*dict*) – A dict structure representing a function and its args and kwargs.

Returns A tuple with the following elements: the name of the function, the args of the function, and the kwargs of the function.

Return type tuple of str, list and dict

insert(*ix*, *func*, **args*, ***kwargs*)
Insert a step to any position in the pipeline.

Args and kwargs here can be of any type. All functions referenced here have to be either imported and passed as function objects or registered with @register_pipeline decorator and passed as function names (strings). For arguments that require calling a different function, use RunnableArgument class.

Parameters

- **func** (*str or function*) – A function or the string name of a function to add to the pipeline.
- **args** (*args*) – Args that are passed to func when calling it.
- **kwargs** (*kwargs*) – Kwargs that are passed to func when calling it.

static is_function(*argument*, *keyword='function'*)
Check if an argument should be converted to a specific object type, e.g. a function or a statement type.

Parameters

- **argument** (*dict or other object*) – The argument is a dict, its keyword entry is checked, and if it is there, we return True, otherwise we return False.
- **keyword** (*Optional[str]*) – The keyword to check if it's there if the argument is a dict. Default: function

run(*statements*, ***kwargs*)
Run all steps of the pipeline.

Parameters

- **statements** (*list[indra.statements.Statement]*) – A list of INDRA Statements to run the pipeline on.
- ****kwargs** (*kwargs*) – It is recommended to define all arguments for the steps functions in the steps definition, but it is also possible to provide some external objects (if it is not possible to provide them as a step argument) as kwargs to the entire pipeline here. One should be cautious to avoid kwargs name clashes between multiple functions (this value will be provided to all functions that expect an argument with the same name). To overwrite this value in other functions, provide it explicitly in the corresponding steps kwargs.

Returns The list of INDRA Statements resulting from running the pipeline on the list of input Statements.

Return type list[indra.statements.Statement]

run_function(*func_dict*, *statements=None*, ***kwargs*)
Run a given function and return the results.

For each of the arguments, if it requires an extra function call, recursively call the functions until we get a simple function.

Parameters

- **func_dict** (*dict*) – A dict representing the function to call, its args and kwargs.
- **args** (*args*) – Args that are passed to the function when calling it.
- **kwargs** (*kwargs*) – Kwargs that are passed to the function when calling it.

Returns Any value that the given function returns.

Return type object

static run_simple_function(*func*, **args*, ***kwargs*)
Run a simple function and return the result.

Simple here means a function all arguments of which are simple values (do not require extra function calls).

Parameters

- **func** (*function*) – The function to call.
- **args** (*args*) – Args that are passed to the function when calling it.
- **kwargs** (*kwargs*) – Kwargs that are passed to the function when calling it.

Returns Any value that the given function returns.

Return type object

to_json_file(*filename*)
Save AssemblyPipeline to a JSON file.

exception indra.pipeline.pipeline.**NotRegisteredFunctionError**
Bases: Exception

class indra.pipeline.pipeline.**RunnableArgument**(*func*, **args*, ***kwargs*)
Bases: object

Class representing arguments generated by calling a function.

RunnableArguments should be used as args or kwargs in AssemblyPipeline *append* and *insert* methods.

Parameters **func** (*str or function*) – A function or a name of a function to be called to generate argument value.

to_json()
Jsonify to standard AssemblyPipeline step format.

indra.pipeline.pipeline.**jsonify_arg_input**(*arg*)
Jsonify user input (in AssemblyPipeline *append* and *insert* methods) into a standard step json.

exception indra.pipeline.decorators.**ExistingFunctionError**
Bases: Exception

indra.pipeline.decorators.**register_pipeline**(*function*)
Decorator to register a function for the assembly pipeline.

4.12 Tools (indra.tools)

4.12.1 Run assembly components in a pipeline (indra.tools.assemble_corpus)

indra.tools.assemble_corpus.**align_statements**(*stmts1*, *stmts2*, *keyfun=None*)
Return alignment of two lists of statements by key.

Parameters

- **stmts1** (*list[indra.statements.Statement]*) – A list of INDRA Statements to align
- **stmts2** (*list[indra.statements.Statement]*) – A list of INDRA Statements to align
- **keyfun** (*Optional[function]*) – A function that takes a Statement as an argument and returns a key to align by. If not given, the default key function is a tuble of the names of the Agents in the Statement.

Returns **matches** – A list of tuples where each tuple has two elements, the first corresponding to an element of the stmts1 list and the second corresponding to an element of the stmts2 list. If a given element is not matched, its corresponding pair in the tuple is None.

Return type list(tuple)

indra.tools.assemble_corpus.**dump_statements**(*stmts_in*, *fname*, *protocol=4*)
Dump a list of statements into a pickle file.

Parameters

- **fname** (*str*) – The name of the pickle file to dump statements into.
- **protocol** (*Optional[int]*) – The pickle protocol to use (use 2 for Python 2 compatibility). Default: 4

indra.tools.assemble_corpus.**dump_stmt_strings**(*stmts*, *fname*)
Save printed statements in a file.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to save in a text file.
- **fname** (*Optional[str]*) – The name of a text file to save the printed statements into.

indra.tools.assemble_corpus.**expand_families**(*stmts_in*, ***kwargs*)
Expand FamPlex Agents to individual genes.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to expand.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of expanded statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_belief**(*stmts_in*, *belief_cutoff*, ***kwargs*)
Filter to statements with belief above a given cutoff.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **belief_cutoff** (*float*) – Only statements with belief above the belief_cutoff will be returned. Here 0 < belief_cutoff < 1.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_by_curation**(*stmts_in*, *curations*, *incorrect_policy='any'*, *correct_tags=None*, *update_belief=True*)
Filter out statements and update beliefs based on curations.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **curations** (*list[dict]*) – A list of curations for evidences. Curation object should have (at least) the following attributes: pa_hash (preassembled statement hash), source_hash (evidence hash) and tag (e.g. 'correct', 'wrong_relation', etc.)
- **incorrect_policy** (*str*) – A policy for filtering out statements given incorrect curations. The 'any' policy filters out a statement if at least one of its evidences is curated as incorrect and no evidences are curated as correct, while the 'all' policy only filters out a statement if all of its evidences are curated as incorrect.
- **correct_tags** (*list[str] or None*) – A list of tags to be considered correct. If no tags are provided, only the 'correct' tag is considered correct.
- **update_belief** (*Option[bool]*) – If True, set a belief score to 1 for statements curated as correct. Default: True

indra.tools.assemble_corpus.**filter_by_db_refs**(*stmts_in*, *namespace*, *values*, *policy*, *invert=False*, *match_suffix=False*, ***kwargs*)
Filter to Statements whose agents are grounded to a matching entry.

Statements are filtered so that the db_refs entry (of the given namespace) of their Agent/Concept arguments take a value in the given list of values.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of Statements to filter.
- **namespace** (*str*) – The namespace in db_refs to which the filter should apply.

- **values** (*list[str]*) – A list of values in the given namespace to which the filter should apply.
- **policy** (*str*) – The policy to apply when filtering for the db_refs. "one": keep Statements that contain at least one of the list of db_refs and possibly others not in the list "all": keep Statements that only contain db_refs given in the list
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.
- **invert** (*Optional[bool]*) – If True, the Statements that do not match according to the policy are returned. Default: False
- **match_suffix** (*Optional[bool]*) – If True, the suffix of the db_refs entry is matches agains the list of entries

Returns **stmts_out** – A list of filtered Statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_by_type**(*stmts_in*, *stmt_type*, *invert=False*, ***kwargs*)
Filter to a given statement type.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **stmt_type** (*str or indra.statements.Statement*) – The class of the statement type to filter for. Alternatively, a string matching the name of the statement class, e.g., "Activation" can be used. Example: indra.statements.Modification or "Modification"
- **invert** (*Optional[bool]*) – If True, the statements that are not of the given type are returned. Default: False
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_complexes_by_size**(*stmts_in*, *members_allowed=5*)
Filter out Complexes if the number of members exceeds specified allowed number.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements from which large Complexes need to be filtered out
- **members_allowed** (*Optional[int]*) – Allowed number of members to include. Default: 5

Returns **stmts_out** – A list of filtered Statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_concept_names**(*stmts_in*, *name_list*, *policy*, *invert=False*, ***kwargs*)
Return Statements that refer to concepts/agents given as a list of names.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of Statements to filter.
- **name_list** (*list[str]*) – A list of concept/agent names to filter for.
- **policy** (*str*) – The policy to apply when filtering for the list of names. "one": keep Statements that contain at least one of the list of names and possibly others not in the list "all": keep Statements that only contain names given in the list

- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.
- **invert** (*Optional[bool]*) – If True, the Statements that do not match according to the policy are returned. Default: False

Returns **stmts_out** – A list of filtered Statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_direct**(*stmts_in*, ***kwargs*)
Filter to statements that are direct interactions

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_enzyme_kinase**(*stmts_in*, ***kwargs*)
Filter Phosphorylations to ones where the enzyme is a known kinase.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_evidence_source**(*stmts_in*, *source_apis*, *policy='one'*, ***kwargs*)
Filter to statements that have evidence from a given set of sources.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **source_apis** (*list[str]*) – A list of sources to filter for. Examples: biopax, bel, reach
- **policy** (*Optional[str]*) – If ‘one’, a statement that hase evidence from any of the sources is kept. If ‘all’, only those statements are kept which have evidence from all the input sources specified in source_apis. If ‘none’, only those statements are kept that don’t have evidence from any of the sources specified in source_apis.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_gene_list**(*stmts_in*, *gene_list*, *policy*, *allow_families=False*, *remove_bound=False*, *invert=False*, ***kwargs*)
Return statements that contain genes given in a list.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **gene_list** (*list[str]*) – A list of gene symbols to filter for.
- **policy** (*str*) – The policy to apply when filtering for the list of genes. “one”: keep statements that contain at least one of the list of genes and possibly others not in the list “all”: keep statements that only contain genes given in the list

- **allow_families** (*Optional[bool]*) – Will include statements involving FamPlex families containing one of the genes in the gene list. Default: False
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.
- **remove_bound** (*Optional[str]*) – If true, removes bound conditions that are not genes in the list If false (default), looks at agents in the bound conditions in addition to those participating in the statement directly when applying the specified policy.
- **invert** (*Optional[bool]*) – If True, the statements that do not match according to the policy are returned. Default: False

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_genes_only**(*stmts_in*, *specific_only=False*, *remove_bound=False*, ***kwargs*)

Filter to statements containing genes only.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **specific_only** (*Optional[bool]*) – If True, only elementary genes/proteins will be kept and families will be filtered out. If False, families are also included in the output. Default: False
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.
- **remove_bound** (*Optional[bool]*) – If true, removes bound conditions that are not genes If false (default), filters out statements with non-gene bound conditions

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_grounded_only**(*stmts_in*, *score_threshold=None*, *remove_bound=False*, ***kwargs*)

Filter to statements that have grounded agents.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **score_threshold** (*Optional[float]*) – If scored groundings are available in a list and the highest score if below this threshold, the Statement is filtered out.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.
- **remove_bound** (*Optional[bool]*) – If true, removes ungrounded bound conditions from a statement. If false (default), filters out statements with ungrounded bound conditions.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_human_only**(*stmts_in*, *remove_bound=False*, ***kwargs*)

Filter out statements that are grounded, but not to a human gene.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

- **remove_bound** (*Optional[bool]*) – If true, removes all bound conditions that are grounded but not to human genes. If false (default), filters out statements with boundary conditions that are grounded to non-human genes.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_inconsequential**(*stmts*, *mods=True*, *mod_whitelist=None*, *acts=True*, *act_whitelist=None*)

Keep filtering inconsequential modifications and activities until there is nothing else to filter.

Parameters

- **stmts** (*list[indra.statements.Statement]*) – A list of INDRA Statements to filter.
- **mods** (*Optional[bool]*) – If True, inconsequential modifications are filtered out. Default: True
- **mod_whitelist** (*Optional[dict]*) – A whitelist containing agent modification sites whose modifications should be preserved even if no other statement refers to them. The whitelist parameter is a dictionary in which the key is a gene name and the value is a list of tuples of (modification_type, residue, position). Example: whitelist = {'MAP2K1': [('phosphorylation', 'S', '222')]}
- **acts** (*Optional[bool]*) – If True, inconsequential activations are filtered out. Default: True
- **act_whitelist** (*Optional[dict]*) – A whitelist containing agent activity types which should be preserved even if no other statement refers to them. The whitelist parameter is a dictionary in which the key is a gene name and the value is a list of activity types. Example: whitelist = {'MAP2K1': ['kinase']}

Returns The filtered list of statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_inconsequential_acts**(*stmts_in*, *whitelist=None*, ***kwargs*)

Filter out Activations that modify inconsequential activities

Inconsequential here means that the site is not mentioned / tested in any other statement. In some cases specific activity types should be preserved, for instance, to be used as readouts in a model. In this case, the given activities can be passed in a whitelist.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **whitelist** (*Optional[dict]*) – A whitelist containing agent activity types which should be preserved even if no other statement refers to them. The whitelist parameter is a dictionary in which the key is a gene name and the value is a list of activity types. Example: whitelist = {'MAP2K1': ['kinase']}
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_inconsequential_mods**(*stmts_in*, *whitelist=None*, ***kwargs*)

Filter out Modifications that modify inconsequential sites

Inconsequential here means that the site is not mentioned / tested in any other statement. In some cases specific sites should be preserved, for instance, to be used as readouts in a model. In this case, the given sites can be passed in a whitelist.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **whitelist** (*Optional[dict]*) – A whitelist containing agent modification sites whose modifications should be preserved even if no other statement refers to them. The whitelist parameter is a dictionary in which the key is a gene name and the value is a list of tuples of (modification_type, residue, position). Example: whitelist = {'MAP2K1': [('phosphorylation', 'S', '222')]}
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_mod_nokinase**(*stmts_in*, ***kwargs*)
Filter non-phospho Modifications to ones with a non-kinase enzyme.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_mutation_status**(*stmts_in*, *mutations*, *deletions*, ***kwargs*)
Filter statements based on existing mutations/deletions

This filter helps to contextualize a set of statements to a given cell type. Given a list of deleted genes, it removes statements that refer to these genes. It also takes a list of mutations and removes statements that refer to mutations not relevant for the given context.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **mutations** (*dict*) – A dictionary whose keys are gene names, and the values are lists of tuples of the form (residue_from, position, residue_to). Example: mutations = {'BRAF': [('V', '600', 'E')]}
- **deletions** (*list*) – A list of gene names that are deleted.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_no_hypothesis**(*stmts_in*, ***kwargs*)
Filter to statements that are not marked as hypothesis in epistemics.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_no_negated**(*stmts_in*, ***kwargs*)
Filter to statements that are not marked as negated in epistemics.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_top_level**(*stmts_in*, ***kwargs*)
Filter to statements that are at the top-level of the hierarchy.

Here top-level statements correspond to most specific ones.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_transcription_factor**(*stmts_in*, ***kwargs*)
Filter out RegulateAmounts where subject is not a transcription factor.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**filter_uuid_list**(*stmts_in*, *uuids*, *invert=True*, ***kwargs*)
Filter to Statements corresponding to given UUIDs

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **uuids** (*list[str]*) – A list of UUIDs to filter for.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.
- **invert** (*Optional[bool]*) – Invert the filter to remove the Statements corresponding to the given UUIDs.

Returns **stmts_out** – A list of filtered statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**fix_invalidities**(*stmts*, *in_place=False*, *print_report_before=False*, *print_report_after=False*, *prior_hash_annots=False*)
Fix invalidities in a list of statements.

Parameters

- **stmts** (List[*Statement*]) – A list of statements to fix invalidities in

- **in_place** (bool) – If True, the statement objects are changed in place if an invalidity is fixed. Otherwise, a deepcopy is done before running fixes.
- **print_report_before** (bool) – Run and print a validation report on the statements before running fixing.
- **print_report_after** (bool) – Run and print a validation report on the statements after running fixing to check if any issues remain that weren't handled by the fixing module.
- **prior_hash_annots** (bool) – If True, an annotation is added to each evidence of a statement with the hash of the statement prior to any fixes being applied. This is useful if this function is applied as a post-processing step on assembled statements and it is necessary to refer back to the original hash of statements before an invalidity fix here potentially changes it. Default: False

Return type List[*Statement*]

Returns The list of statements with validation issues fixed and some invalid statements filtered out.

indra.tools.assemble_corpus.**load_statements**(*fname*, *as_dict=False*)

Load statements from a pickle file.

Parameters

- **fname** (*str*) – The name of the pickle file to load statements from.
- **as_dict** (*Optional[bool]*) – If True and the pickle file contains a dictionary of statements, it is returned as a dictionary. If False, the statements are always returned in a list. Default: False

Returns **stmts** – A list or dict of statements that were loaded.

Return type list

indra.tools.assemble_corpus.**map_db_refs**(*stmts_in*, *db_refs_map=None*)

Update entries in db_refs to those provided in db_refs_map.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of INDRA Statements to update db_refs in.
- **db_refs_map** (*Optional[dict]*) – A dictionary where each key is a tuple (db_ns, db_id) representing old db_refs pair that has to be updated and each value is a new db_id to replace the old value with. If not provided, the default db_refs_map will be loaded.

indra.tools.assemble_corpus.**map_grounding**(*stmts_in*, *do_rename=True*, *grounding_map=None*, *misgrounding_map=None*, *agent_map=None*, *ignores=None*, *use_adeft=True*, *gilda_mode=None*, *grounding_map_policy='replace'*, ***kwargs*)

Map grounding using the GroundingMapper.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to map.
- **do_rename** (*Optional[bool]*) – If True, Agents are renamed based on their mapped grounding.
- **grounding_map** (*Optional[dict]*) – A user supplied grounding map which maps a string to a dictionary of database IDs (in the format used by Agents' db_refs).
- **misgrounding_map** (*Optional[dict]*) – A user supplied misgrounding map which maps a string to a known misgrounding which can be eliminated by the grounding mapper.

- **ignores** (*Optional[list]*) – A user supplied list of ignorable strings which, if present as an Agent text in a Statement, the Statement is filtered out.
- **use_adeft** (*Optional[bool]*) – If True, Adeft will be attempted to be used for acronym disambiguation. Default: True
- **gilda_mode** (*Optional[str]*) – If None, Gilda will not be for disambiguation. If 'web', the address set in the GILDA_URL configuration or environmental variable is used as a Gilda web service. If 'local', the gilda package is imported and used locally.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.
- **grounding_map_policy** (*Optional[str]*) – If a grounding map is provided, use the policy to extend or replace a default grounding map. Default: 'replace'.

Returns **stmts_out** – A list of mapped statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**map_sequence**(*stmts_in*, *do_methionine_offset=True*, *do_orthology_mapping=True*, *do_isoform_mapping=True*, ***kwargs*)

Map sequences using the SiteMapper.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to map.
- **do_methionine_offset** (*boolean*) – Whether to check for off-by-one errors in site position (possibly) attributable to site numbering from mature proteins after cleavage of the initial methionine. If True, checks the reference sequence for a known modification at 1 site position greater than the given one; if there exists such a site, creates the mapping. Default is True.
- **do_orthology_mapping** (*boolean*) – Whether to check sequence positions for known modification sites in mouse or rat sequences (based on PhosphoSitePlus data). If a mouse/rat site is found that is linked to a site in the human reference sequence, a mapping is created. Default is True.
- **do_isoform_mapping** (*boolean*) – Whether to check sequence positions for known modifications in other human isoforms of the protein (based on PhosphoSitePlus data). If a site is found that is linked to a site in the human reference sequence, a mapping is created. Default is True.
- **use_cache** (*boolean*) – If True, a cache will be created/used from the laction specified by SITEMAPPER_CACHE_PATH, defined in your INDRA config or the environment. If False, no cache is used. For more details on the cache, see the SiteMapper class definition.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of mapped statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**merge_groundings**(*stmts_in*)

Gather and merge original grounding information from evidences.

Each Statement's evidences are traversed to find original grounding information. These groundings are then merged into an overall consensus grounding dict with as much detail as possible.

The current implementation is only applicable to Statements whose concept/agent roles are fixed. Complexes, Associations and Conversions cannot be handled correctly.

Parameters **stmts_in** (*list[indra.statements.Statement]*) – A list of INDRA Statements whose groundings should be merged. These Statements are meant to have been preassembled and potentially have multiple pieces of evidence.

Returns **stmts_out** – The list of Statements now with groundings merged at the Statement level.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**normalize_active_forms**(*stmts_in*)
Run preassembly of ActiveForms only and keep other statements unchanged.

This is specifically useful in the special case of mechanism linking (that is run after preassembly) producing ActiveForm statements that are redundant. Otherwise, general preassembly deduplicates ActiveForms as expected.

Parameters **stmts_in** (*list[indra.statements.Statement]*) – A list of INDRA Statements among which ActiveForms should be normalized.

Returns A list of INDRA Statements in which ActiveForms are normalized.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**reduce_activities**(*stmts_in*, ***kwargs*)
Reduce the activity types in a list of statements

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to reduce activity types in.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of reduced activity statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**rename_db_ref**(*stmts_in*, *ns_from*, *ns_to*, ***kwargs*)
Rename an entry in the db_refs of each Agent.

This is particularly useful when old Statements in pickle files need to be updated after a namespace was changed such as 'BE' to 'FPLX'.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements whose Agents' db_refs need to be changed
- **ns_from** (*str*) – The namespace identifier to replace
- **ns_to** (*str*) – The namespace identifier to replace to
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of Statements with Agents' db_refs changed.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**run_mechlinker**(*stmts_in*, *reduce_activities=False*, *reduce_modifications=False*, *replace_activations=False*, *require_active_forms=False*, *implicit=False*)
Instantiate MechLinker and run its methods in defined order.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of INDRA Statements to run mechanism linking on.

- **reduce_activities** (*Optional[bool]*) – If True, agent activities are reduced to their most specific, unambiguous form. Default: False
- **reduce_modifications** (*Optional[bool]*) – If True, agent modifications are reduced to their most specific, unambiguous form. Default: False
- **replace_activations** (*Optional[bool]*) – If True, if there is compatible pair of Modification(X, Y) and ActiveForm(Y) statements, then any Activation(X,Y) statements are filtered out. Default: False
- **require_active_forms** (*Optional[bool]*) – If True, agents in active positions are rewritten to be in their active forms. Default: False
- **implicit** (*Optional[bool]*) – If True, active forms of an agent are inferred from multiple statement types implicitly, otherwise only explicit ActiveForm statements are taken into account. Default: False

Returns A list of INDRA Statements that have gone through mechanism linking.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**run_preassembly**(*stmts_in*, *return_toplevel=True*, *poolsize=None*, *size_cutoff=None*, *belief_scorer=None*, *ontology=None*, *matches_fun=None*, *refinement_fun=None*, *flatten_evidence=False*, *flatten_evidence_collect_from=None*, *normalize_equivalences=False*, *normalize_opposites=False*, *normalize_ns='WM'*, *run_refinement=True*, *filters=None*, ***kwargs*)

Run preassembly on a list of statements.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to preassemble.
- **return_toplevel** (*Optional[bool]*) – If True, only the top-level statements are returned. If False, all statements are returned irrespective of level of specificity. Default: True
- **poolsize** (*Optional[int]*) – The number of worker processes to use to parallelize the comparisons performed by the function. If None (default), no parallelization is performed. NOTE: Parallelization is only available on Python 3.4 and above.
- **size_cutoff** (*Optional[int]*) – Groups with size_cutoff or more statements are sent to worker processes, while smaller groups are compared in the parent process. Default value is 100. Not relevant when parallelization is not used.
- **belief_scorer** (*Optional[indra.belief.BeliefScorer]*) – Instance of BeliefScorer class to use in calculating Statement probabilities. If None is provided (default), then the default scorer is used.
- **ontology** (*Optional[IndraOntology]*) – IndraOntology object to use for preassembly
- **matches_fun** (*Optional[function]*) – A function to override the built-in matches_key function of statements.
- **refinement_fun** (*Optional[function]*) – A function to override the built-in refinement_of function of statements.
- **flatten_evidence** (*Optional[bool]*) – If True, evidences are collected and flattened via supports/supported_by links. Default: False

- **flatten_evidence_collect_from** (*Optional[str]*) – String indicating whether to collect and flatten evidence from the *supports* attribute of each statement or the *supported_by* attribute. If not set, defaults to 'supported_by'. Only relevant when flatten_evidence is True.
- **normalize_equivalences** (*Optional[bool]*) – If True, equivalent groundings are rewritten to a single standard one. Default: False
- **normalize_opposites** (*Optional[bool]*) – If True, groundings that have opposites in the ontology are rewritten to a single standard one.
- **normalize_ns** (*Optional[str]*) – The name space with respect to which equivalences and opposites are normalized.
- **filters** (*Optional[list[:py:class:indra.preassembler.refinement.RefinementFilter]]*) – A list of RefinementFilter classes that implement filters on possible statement refinements. For details on how to construct such a filter, see the documentation of *indra.preassembler.refinement.RefinementFilter*. If no user-supplied filters are provided, the default ontology-based filter is applied. If a list of filters is provided here, the *indra.preassembler.refinement.OntologyRefinementFilter* isn't appended by default, and should be added by the user, if necessary. Default: None
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.
- **save_unique** (*Optional[str]*) – The name of a pickle file to save the unique statements into.

Returns **stmts_out** – A list of preassembled top-level statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**run_preassembly_duplicate**(*preassembler*, *beliefengine*, ***kwargs*)
Run deduplication stage of preassembly on a list of statements.

Parameters

- **preassembler** (indra.preassembler.Preassembler) – A Preassembler instance
- **beliefengine** (indra.belief.BeliefEngine) – A BeliefEngine instance.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of unique statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**run_preassembly_related**(*preassembler*, *beliefengine*, ***kwargs*)
Run related stage of preassembly on a list of statements.

Parameters

- **preassembler** (indra.preassembler.Preassembler) – A Preassembler instance which already has a set of unique statements internally.
- **beliefengine** (indra.belief.BeliefEngine) – A BeliefEngine instance.
- **return_toplevel** (*Optional[bool]*) – If True, only the top-level statements are returned. If False, all statements are returned irrespective of level of specificity. Default: True
- **size_cutoff** (*Optional[int]*) – Groups with size_cutoff or more statements are sent to worker processes, while smaller groups are compared in the parent process. Default value is 100. Not relevant when parallelization is not used.

- **flatten_evidence** (*Optional[bool]*) – If True, evidences are collected and flattened via supports/supported_by links. Default: False
- **flatten_evidence_collect_from** (*Optional[str]*) – String indicating whether to collect and flatten evidence from the *supports* attribute of each statement or the *supported_by* attribute. If not set, defaults to 'supported_by'. Only relevant when flatten_evidence is True.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of preassembled top-level statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**standardize_names_groundings**(*stmts*)
Standardize the names of Concepts with respect to an ontology.

NOTE: this function is currently optimized for Influence Statements obtained from Eidos, Hume, Sofia and CWMS. It will possibly yield unexpected results for biology-specific Statements.

Parameters **stmts** (*list[indra.statements.Statement]*) – A list of statements whose Concept names should be standardized.

indra.tools.assemble_corpus.**strip_agent_context**(*stmts_in*, ***kwargs*)
Strip any context on agents within each statement.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements whose agent context should be stripped.
- **save** (*Optional[str]*) – The name of a pickle file to save the results (stmts_out) into.

Returns **stmts_out** – A list of stripped statements.

Return type list[indra.statements.Statement]

indra.tools.assemble_corpus.**strip_supports**(*stmts*)
Remove supports and supported by from statements.

4.12.2 Fix common invalidities in Statements (indra.tools.fix_invalidities)

indra.tools.fix_invalidities.**fix_invalidities**(*stmts*)
Fix invalidities in a list of Statements.

Note that in some cases statements can be filtered out if there is a known issue to which there is no fix, e.g., a Translocation statements missing both location parameters.

Parameters **stmts** (List[*Statement*]) – A list of INDRA Statements.

Return type List[*Statement*]

Returns The list of statements with invalidities fixed.

indra.tools.fix_invalidities.**fix_invalidities_agent**(*agent*)
Fix invalidities of a single INDRA Agent in place.

indra.tools.fix_invalidities.**fix_invalidities_context**(*context*)
Fix invalidities of a single INDRA BioContext in place.

indra.tools.fix_invalidities.**fix_invalidities_db_refs**(*db_refs*)
Return a fixed version of a db_refs grounding dict.

Return type Mapping[str, str]

indra.tools.fix_invalidities.**fix_invalidities_evidence**(*ev*)
Fix invalidities of a single INDRA Evidence in place.

indra.tools.fix_invalidities.**fix_invalidities_stmt**(*stmt*)
Fix invalidities of a single INDRA Statement in place.

4.12.3 Annotate websites with INDRA through hypothes.is (indra.tools.hypothesis_annotator)

This module exposes functions that annotate websites (including PubMed and PubMedCentral pages, or any other text-based website) with INDRA Statements through hypothes.is. Features include reading the content of the website 'de-novo', and generating new INDRA Statements for annotation, and fetching existing statements for a paper from the INDRA DB and using those for annotation.

indra.tools.hypothesis_annotator.**annotate_paper_from_db**(*text_refs*, *assembly_pipeline=None*)
Upload INDRA Statements as annotations for a given paper based on content for that paper in the INDRA DB.

Parameters

- **text_refs** (*dict*) – A dict of text references, following the same format as the INDRA Evidence text_refs attribute.
- **assembly_pipeline** (*Optional[json]*) – A list of pipeline steps (typically filters) that are applied before uploading statements to hypothes.is as annotations.

indra.tools.hypothesis_annotator.**read_and_annotate**(*text_refs*, *text_extractor=None*, *text_reader=None*, *assembly_pipeline=None*)
Read a paper/website and upload annotations derived from it to hypothes.is.

Parameters

- **text_refs** (*dict*) – A dict of text references, following the same format as the INDRA Evidence text_refs attribute.
- **text_extractor** (*Optional[function]*) – A function which takes the raw content of a website (e.g., HTML) and extracts clean text from it to prepare for machine reading. This is only used if the text_refs is a URL (e.g., a Wikipedia page), it is not used for PMID or PMCID text_refs where content can be pre-processed and machine read directly. Default: None Example: html2text.HTML2Text().handle
- **text_reader** (*Optional[function]*) – A function which takes a single text string argument (the text extracted from a given resource), runs reading on it, and returns a list of INDRA Statement objects. Due to complications with the PMC NXML format, this option only supports URL or PMID resources as input in text_refs. Default: None. In the default case, the INDRA REST API is called with an appropriate endpoint that runs Reach and processes its output into INDRA Statements.
- **assembly_pipeline** (*Optional[json]*) – A list of assembly pipeline steps that are applied before uploading statements to hypothes.is as annotations. Example: [{'function': 'map_grounding'}]

4.12.4 Build a network from a gene list (indra.tools.gene_network)

class indra.tools.gene_network.**GeneNetwork**(*gene_list*, *basename=None*)

Build a set of INDRA statements for a given gene list from databases.

Parameters

- **gene_list** (*list[str]*) – List of gene names.
- **basename** (*str or None (default)*) – Filename prefix to be used for caching of intermediates (Biopax OWL file, pickled statement lists, etc.). If None, no results are cached and no cached files are used.

gene_list

List of gene names

Type list[str]

basename

Filename prefix for cached intermediates, or None if no cached used.

Type str or None

results

List of preassembled statements.

Type list[indra.statements.Statement]

get_bel_stmts(*filter=False*)

Get relevant statements from the BEL large corpus.

Performs a series of neighborhood queries and then takes the union of all the statements. Because the query process can take a long time for large gene lists, the resulting list of statements are cached in a pickle file with the filename *<basename>_bel_stmts.pkl*. If the pickle file is present, it is used by default; if not present, the queries are performed and the results are cached.

Parameters **filter** (*bool*) – If True, includes only those statements that exclusively mention genes in *gene_list*. Default is False. Note that the full (unfiltered) set of statements are cached.

Returns List of INDRA statements extracted from the BEL large corpus.

Return type list of indra.statements.Statement

get_biopax_stmts(*filter=False*, *query='pathsbetween'*, *database_filter=None*)

Get relevant statements from Pathway Commons.

Performs a "paths between" query for the genes in *gene_list* and uses the results to build statements. This function caches two files: the list of statements built from the query, which is cached in *<basename>_biopax_stmts.pkl*, and the OWL file returned by the Pathway Commons Web API, which is cached in *<basename>_pc_pathsbetween.owl*. If these cached files are found, then the results are returned based on the cached file and Pathway Commons is not queried again.

Parameters

- **filter** (*Optional[bool]*) – If True, includes only those statements that exclusively mention genes in *gene_list*. Default is False.
- **query** (*Optional[str]*) – Defined what type of query is executed. The two options are 'pathsbetween' which finds paths between the given list of genes and only works if more than 1 gene is given, and 'neighborhood' which searches the immediate neighborhood of each given gene. Note that for pathsbetween queries with more thatn 60 genes, the query will be executed in multiple blocks for scalability.

- **database_filter** (*Optional[list[str]]*) – A list of PathwayCommons databases to include in the query.

Returns List of INDRA statements extracted from Pathway Commons.

Return type list of `indra.statements.Statement`

`get_statements`(*filter=False*)

Return the combined list of statements from BEL and Pathway Commons.

Internally calls *get_biopax_stmts()* and *get_bel_stmts()*.

Parameters **filter** (*bool*) – If True, includes only those statements that exclusively mention genes in *gene_list*. Default is False.

Returns List of INDRA statements extracted the BEL large corpus and Pathway Commons.

Return type list of `indra.statements.Statement`

`run_preassembly`(*stmts*, *print_summary=True*)

Run complete preassembly procedure on the given statements.

Results are returned as a dict and stored in the attribute *results*. They are also saved in the pickle file *<basename>_results.pkl*.

Parameters

- **stmts** (list of `indra.statements.Statement`) – Statements to preassemble.
- **print_summary** (*bool*) – If True (default), prints a summary of the preassembly process to the console.

Returns

A dict containing the following entries:

- *raw*: the starting set of statements before preassembly.
- *duplicates1*: statements after initial de-duplication.
- *valid*: statements found to have valid modification sites.
- *mapped*: mapped statements (list of *indra.preassembler.sitemapper.MappedStatement*).
- *mapped_stmts*: combined list of valid statements and statements after mapping.
- *duplicates2*: statements resulting from de-duplication of the statements in *mapped_stmts*.
- *related2*: top-level statements after combining the statements in *duplicates2*.

Return type dict

4.12.5 Build an executable model from a fragment of a large network (`indra.tools.executable_subnetwork`)

`indra.tools.executable_subnetwork.get_subnetwork`(*statements*, *nodes*)

Return a PySB model based on a subset of given INDRA Statements.

Statements are first filtered for nodes in the given list and other nodes are optionally added based on relevance in a given network. The filtered statements are then assembled into an executable model using INDRA's PySB Assembler.

Parameters

- **statements** (*list[indra.statements.Statement]*) – A list of INDRA Statements to extract a subnetwork from.
- **nodes** (*list[str]*) – The names of the nodes to extract the subnetwork for.

Returns **model** – A PySB model object assembled using INDRA's PySB Assembler from the INDRA Statements corresponding to the subnetwork.

Return type pysb.Model

4.12.6 Build a model incrementally over time (indra.tools.incremental_model)

class indra.tools.incremental_model.**IncrementalModel**(*model_fname=None*)

Assemble a model incrementally by iteratively adding new Statements.

Parameters **model_fname** (*Optional[str]*) – The name of the pickle file in which a set of INDRA Statements are stored in a dict keyed by PubMed IDs. This is the state of an IncrementalModel that is loaded upon instantiation.

stmts

A dictionary of INDRA Statements keyed by PMIDs that stores the current state of the IncrementalModel.

Type dict[str, list[indra.statements.Statement]]

assembled_stmts

A list of INDRA Statements after assembly.

Type list[indra.statements.Statement]

add_statements(*pmid*, *stmts*)

Add INDRA Statements to the incremental model indexed by PMID.

Parameters

- **pmid** (*str*) – The PMID of the paper from which statements were extracted.
- **stmts** (*list[indra.statements.Statement]*) – A list of INDRA Statements to be added to the model.

get_model_agents()

Return a list of all Agents from all Statements.

Returns **agents** – A list of Agents that are in the model.

Return type list[indra.statements.Agent]

get_statements()

Return a list of all Statements in a single list.

Returns **stmts** – A list of all the INDRA Statements in the model.

Return type list[indra.statements.Statement]

get_statements_noprior()

Return a list of all non-prior Statements in a single list.

Returns **stmts** – A list of all the INDRA Statements in the model (excluding the prior).

Return type list[indra.statements.Statement]

get_statements_prior()

Return a list of all prior Statements in a single list.

Returns **stmts** – A list of all the INDRA Statements in the prior.

Return type list[indra.statements.Statement]

load_prior(*prior_fname*)
Load a set of prior statements from a pickle file.

The prior statements have a special key in the stmts dictionary called "prior".

Parameters **prior_fname** (*str*) – The name of the pickle file containing the prior Statements.

preassemble(*filters=None*, *grounding_map=None*)
Preassemble the Statements collected in the model.

Use INDRA's GroundingMapper, Preassembler and BeliefEngine on the IncrementalModel and save the unique statements and the top level statements in class attributes.

Currently the following filter options are implemented: - grounding: require that all Agents in statements are grounded - human_only: require that all proteins are human proteins - prior_one: require that at least one Agent is in the prior model - prior_all: require that all Agents are in the prior model

Parameters

- **filters** (*Optional[list[str]]*) – A list of filter options to apply when choosing the statements. See description above for more details. Default: None
- **grounding_map** (*Optional[dict]*) – A user supplied grounding map which maps a string to a dictionary of database IDs (in the format used by Agents' db_refs).

save(*model_fname='model.pkl'*)
Save the state of the IncrementalModel in a pickle file.

Parameters **model_fname** (*Optional[str]*) – The name of the pickle file to save the state of the IncrementalModel in. Default: model.pkl

4.12.7 The RAS Machine (indra.tools.machine)

Prerequisites

First, install the machine-specific dependencies:

```
pip install indra[machine]
```

Starting a New Model

To start a new model, run

```
python -m indra.tools.machine make model_name
```

Alternatively, the command line interface can be invoked with

```
indra-machine make model_name
```

where model_name corresponds to the name of the model to initialize.

This script generates the following folders and files

- model_name
- model_name/log.txt
- model_name/config.yaml

- model_name/jsons/

You should the edit model_name/config.yaml to set up the search terms and optionally the credentials to use Twitter, Gmail or NDEx bindings.

Setting Up Search Terms

The config.yml file is a standard YAML configuration file. A template is available in model_name/config.yaml after having created the machine.

Two important fields in config.yml are `search_terms` and `search_genes` both of which are YAML lists. The entries of `search_terms` are used _directly_ as queries in PubMed search (for more information on PubMed search strings, read https://www.ncbi.nlm.nih.gov/books/NBK3827/#pubmedhelp.Searching_PubMed).

Example:

```
search_terms:
- breast cancer
- proteasome
- apoptosis
```

The entries of `search_genes` is a special list in which _only_ standard HGNC gene symbols are allowed. Entries in this list are also used to search PubMed but also serve as a list of *prior* genes that are known to be relevant for the model.

#Entries in this can be used to search #PubMed specifically for articles that are tagged with the gene's unique #identifier rather than its string name. This mode of searching for articles #on specific genes is much more reliable than searching for them using #string names.

Example:

```
search_genes:
- AKT1
- MAPK3
- EGFR
```

Extending a Model

To extend a model, run

```
python -m indra.tools.machine run_with_search model_name
```

Alternatively, the command line interface can be invoked with

```
indra-machine run_with_search model_name
```

Extending a model involves extracting PMIDs from emails (if Gmail credentials are given), and searching using INDRA's PubMed client with each entry of search_terms in config.yaml as a search term. INDRA's literature client is then used to find the full text corresponding to each PMID or its abstract when the full text is not available. The REACH parser is then used to read each new paper. INDRA uses the REACH output to construct Statements corresponding to mechanisms. It then adds them to an incremental model through a process of assembly involving duplication and overlap resolution and the application of filters.

`indra.tools.machine.`**`copy_default_config`**(*destination*)
Copies the default configuration to the given destination

Parameters **destination** (*str*) – The location to which a default RAS Machine config file is placed.

4.13 Resource files

This module contains a number of resource files that INDRA uses to perform tasks such as name standardization and ID mapping.

indra.resources.**get_resource_path**(*fname*)
Return the absolute path to a file in the resource folder.

indra.resources.**load_resource_json**(*fname*)
Load a given JSON file from the resources folder.

Parameters **fname** (*str*) – The name of the json file in the resources folder.

Returns The content of the JSON file loaded into a dict/list.

Return type json

indra.resources.**open_resource_file**(*resource_name*, **args*, ***kwargs*)
Return a file handle to an INDRA resource file.

4.14 Util (indra.util)

4.14.1 Statement presentation (indra.util.statement_presentation)

This module groups and sorts Statements for presentation in downstream tools while aggregating the statements' statistics/metrics into the groupings. While most usage of this module will be via the top-level function *group_and_sort_statements*, alternative usages (including custom statement data, multiple statement grouping levels, and multiple strategies for aggregating statement-level metrics for higher-level groupings) are supported through the various classes (see Class Overview below).

Vocabulary

An "agent-pair" is, as the name suggests, a pair of agents from a statement, usually defined by their canonical names.

A "relation" is the basic information of a statement, with all details (such as sites, residues, mutations, and bound conditions) stripped away. Usually this means it is just the statement type (or verb), subject name, and object name, though in some corner cases it is different.

Simple Example

The principal function in the module is *group_and_sort_statements*, and if you want statements grouped into agent-pairs, then by relations, sorted by evidence count, simply use the function with its defaults, e.g.:

```
for _, ag_key, rels, ag_metrics in group_and_sort_statements(stmts):
    print(ag_key)
    for _, rel_key, stmt_data, rel_metrics in rels:
        print('       ', rel_key)
        for _, stmt_hash, stmt_obj, stmt_metrics in stmt_data:
            print('           ', stmt_obj)
```

Advanced Example

Custom data and aggregation methods are supported, respectively, by using instances of the *StmtStat* class and subclassing the BasicAggregator (or more generally, the AggregatorMeta) API. Custom sorting is implemented by defining and passing a *sort_by* function to *group_and_sort_statements*.

For example, if you have custom statement metrics (e.g., a value obtained by experiment such as differential expression of subject or object genes), want the statements grouped only to the level of relations, and want to sort the statements and relations independently. Suppose also that your measurement applies equally at the statement and relation level and hence you don't want any changes applied during aggregation (e.g. averaging). This is illustrated in the example below:

```
# Define a new aggregator that doesn't apply any aggregation function to
# the data, simply taking the last metric (effectively a noop):
class NoopAggregator(BasicAggregator):
    def _merge(self, metric_array):
        self.values = metric_array

# Create your StmtStat using custom data dict `my_data`, a dict of values
# keyed by statement hash:
my_stat = StmtStat('my_stat', my_data, int, NoopAggregator)

# Define a custom sort function using my stat and the default available
# ev_count. In effect this will sort relations by the custom stat, and then
# secondarily sort the statements within that relation (for which my_stat
# is by design the same) using their evidence counts.
def my_sort(metrics):
    return metrics['my_stat'], metrics['ev_count']

# Iterate over the results.
groups = group_and_sort_statements(stmts, sort_by=my_sort,
                                   custom_stats=[my_stat],
                                   grouping_level='relation')
for _, rel_key, rel_stmts, rel_metrics in groups:
    print(rel_key, rel_metrics['my_stat'])
    for _, stmt_hash, stmt, metrics in rel_stmts:
        print('      ', stmt, metrics['ev_count'])
```

Class Overview

Statements can have multiple metrics associated with them, most commonly belief, evidence counts, and source counts, although other metrics may also be applied. Such metrics imply an order on the set of Statements, and a user should be able to apply that order to them for sorting or filtering. them. These types of metric, or "stat", are represented by *StmtStat* classes.

Statements can be grouped based on the information they represent: by their agents (e.g. subject is MEK and object is ERK), and by their type (e.g. Phosphorylation). These groups are represented by *StmtGroup* objects, which on their surface behave much like *defaultdict(list)* would, though more is going on behind the scenes. The StmtGroup class is used internally by *group_and_sort_statements* and would only need to be used directly if defining an alternative statement-level grouping approach (e.g., grouping statements by subject).

Like Statements, higher-level statement groups are subject to sorting and filtering. That requires that the *StmtStat`s be aggregated over the statements in a group. The Aggregator classes serve this purpose, using numpy to do sums over arrays of metrics as Statements are "included" in the `StmtGroup*. Each *StmtStat* must declare how its data should

be aggregated, as different kinds of data aggregate differently. Custom aggregation methods can be implemented by subclassing the *BasicAggregator* class and using an instance of the custom class to define a *StmtStat*.

class indra.util.statement_presentation.**AggregatorMeta**
Define the API for an aggregator of statement metrics.

In general, an aggregator defines the ways that different kinds of statement metrics are merged into groups. For example, evidence counts are aggregated by summing, as are counts for various sources. Beliefs are aggregated over a group of statements by maximum (usually).

get_dict()
Get a dictionary representation of the data in this aggregate.

Keys are those originally given to the StmtStat instances used to build this aggregator.

include(*stmt*)
Add the metrics from the given statement to this aggregate.

class indra.util.statement_presentation.**AveAggregator**(*keys*, *stmt_metrics*, *original_types*)
A stats aggregator averages the included statement metrics.

class indra.util.statement_presentation.**BasicAggregator**(*keys*, *stmt_metrics*, *original_types*)
Gathers measurements for a statement or similar entity.

By defining a child of BasicAggregator, specifically defining the operations that gather new data and finalize that data once all the statements are collected, one can use arbitrary statistical methods to aggregate metrics for high-level groupings of Statements for subsequent sorting or filtering purposes.

Parameters

- **keys** (*list[str]*) – A dict keyed by aggregation method of lists of the names for the elements of data.
- **stmt_metrics** (*dict{int: np.ndarray}*) – A dictionary keyed by hash with each element a dict of arrays keyed by aggregation type.
- **original_types** (*tuple(type)*) – The type classes of each numerical value stored in the base_group dict, e.g. *(int, float, int)*.

get_dict()
Get a dictionary representation of the data in this aggregate.

Keys are those originally given to the StmtStat instances used to build this aggregator.

include(*stmt*)
Include a statement and its statistics in the group.

class indra.util.statement_presentation.**MaxAggregator**(*keys*, *stmt_metrics*, *original_types*)
A stats aggregator that takes the max of statement metrics.

class indra.util.statement_presentation.**MultiAggregator**(*basic_aggs*)
Implement the AggregatorMeta API for multiple BasicAggregator children.

Takes an iterable of BasicAggregator children.

get_dict()
Get a dictionary representation of the data in this aggregate.

Keys are those originally given to the StmtStat instances used to build this aggregator.

include(*stmt*)
Add the metrics from the given statement to this aggregate.

class indra.util.statement_presentation.**StmtGroup**(*stat_groups*)

Creates higher-level stmt groupings and aggregates metrics accordingly.

Used internally by *group_and_sort_statements*.

This class manages the accumulation of statistics for statement groupings, such as by relation or agent pair. It calculates metrics for these higher-level groupings using metric-specific aggregators implementing the AggregatorMeta API (e.g., MultiAggregator and any children of BasicAggregator).

For example, evidence counts for a relation can be calculated as the sum of the statement-level evidence counts, while the belief for the relation can be calculated as the average or maximum of the statement-level beliefs.

The primary methods for instantiating this class are the two factory class methods: - from_stmt_stats - from_dicts See the methods for more details on their purpose and usage.

Once instantiated, the StmtGroup behaves like a defaultdict of lists, where the keys are group-level keys, and the lists contain statements. Statements can be iteratively added to the group via the dict-like syntax *stmt_group[group_key].include(stmt)*. This allows the caller to generate keys and trigger metric aggregation in a single iteration over statements.

Example usage:

```
# Get ev_count, belief, and ag_count from a list of statements.
stmt_stats = StmtStat.from_stmts(stmt_list)

# Add another stat for a measure of relevance
stmt_stats.append(
    StmtStat('relevance', relevance_dict, float, AveAggregator)
)

# Create the Group
sg = StmtGroup.from_stmt_stats(*stmt_stats)

# Load it full of Statements, grouped by agents.
sg.fill_from_stmt_stats()
sg.start()
for s in stmt_list:
   key = (ag.get_grounding() for ag in s.agent_list())
   sg[key].include(s)
sg.finish()

# Now the stats for each group are aggregated and available for use.
metrics = sg[(('FPLX', 'MEK'), ('FPLX', 'ERK'))].get_dict()
```

add_stats(**stmt_stats*)

Add more stats to the object.

If you have started accumulating data from statements and doing aggregation, (e.g. if you have "started"), or if you are "finished", this request will lead to an error.

fill_from_stmt_stats()

Use the statements stats as stats and hashes as keys.

This is used if you decide you just want to represent statements.

finish()

Finish adding entries, new keys will be rejected.

classmethod from_dicts(*ev_counts=None*, *beliefs=None*, *source_counts=None*)

Init a stmt group from dicts keyed by hash.

Return a StmtGroup constructed from the given keyword arguments. The dict keys of *source_counts* will be broken out into their own StmtStat objects, so that the resulting data model is in effect a flat list of measurement parameters. There is some risk of name collision, so take care not to name any sources "ev_counts" or "belief".

classmethod from_stmt_stats(**stmt_stats*)
Create a stmt group from StmtStat objects.

Return a StmtGroup constructed existing StmtStat objects. This method offers the user the most control and customizability.

get_new_instance()
Create an instance to gather another level of data.

row_set()
Get a set of the rows (data labels) of the stats in this instance.

start()
Mark the start of Statement aggregation.

This will freeze the addition of StmtStats and will enable new keyed entries to be added and aggregated.

class indra.util.statement_presentation.**StmtStat**(*name*, *data*, *data_type*, *agg_class*)
Abstraction of a metric applied to a set of statements.

Can be instantiated either via the constructor or two factory class methods: - s = StmtStat(name, {hash: value, ...}, data_type, AggClass) - [s1, ...] = StmtStat.from_dicts({hash: {label: value, ...}, ...}, data_type, AggClass) - [s_ev_count, s_belief] = StmtStat.from_stmts([Statement(), ...], ('ev_count', 'belief'))

Note that each stat will have only one metric associated with it, so dicts ingested by *from_dicts* will have their values broken up into separate StmtStat instances.

Parameters

- **name** (*str*) – The label for this data (e.g. "ev_count" or "belief")
- **data** (*dict{int: Number}*) – The relevant statistics as a dict keyed by hash.
- **data_type** (*type*) – The type of the data (e.g. *int* or *float*).
- **agg_class** (*type*) – A subclass of BasicAggregator which defines how these statistics will be merged.

classmethod from_dicts(*dict_data*, *data_type*, *agg_class*)
Generate a list of StmtStat's from a dict of dicts.

Example Usage: >> source_counts = {9623812756876: {'reach': 1, 'sparser': 2}, >> -39877587165298: {'reach': 3, 'sparser': 0}} >> stmt_stats = StmtStat.from_dicts(source_counts, int, SumAggregator)

Parameters

- **dict_data** (*dict{int: dict{str: Number}}*) – A dictionary keyed by hash with dictionary elements, where each element gives a set of measurements for the statement labels as keys. A common example is *source_counts*.
- **data_type** (*type*) – The type of the data being given (e.g. *int* or *float*).
- **agg_class** (*type*) – A subclass of BasicAggregator which defines how these statistics will be merged (e.g. *SumAggregator*).

classmethod from_stmts(*stmt_list*, *values=None*)
Generate a list of StmtStat's from a list of stmts.

The stats will include "ev_count", "belief", and "ag_count" by default, but a more limited selection may be specified using *values*.

Example usage: >> stmt_stats = StmtStat.from_stmts(stmt_list, ('ag_count', 'belief'))

Parameters

- **stmt_list** (*list[Statement]*) – A list of INDRA statements, from which basic stats will be derived.
- **values** (*Optional[tuple(str)]*) – A tuple of the names of the values to gather from the list of statements. For example, if you already have evidence counts, you might only want to gather belief and agent counts.

class indra.util.statement_presentation.**SumAggregator**(*keys*, *stmt_metrics*, *original_types*)
A stats aggregator that executes a sum.

indra.util.statement_presentation.**all_sources = ['psp', 'cbn', 'pc', 'bel_lc', 'signor', 'biogrid', 'tas', 'hprd', 'trrust', 'ctd', 'vhn', 'pe', 'drugbank', 'omnipath', 'conib', 'crog', 'dgi', 'minerva', 'creeds', 'ubibrowser', 'acsn', 'geneways', 'tees', 'gnbr', 'isi', 'trips', 'rlimsp', 'medscan', 'eidos', 'sparser', 'reach']**
Source names as they appear in the DB

indra.util.statement_presentation.**available_sources_src_counts**(*source_counts*, *custom_sources=None*)
Returns the set of sources available from a source counts dict

Return type Set[str]

indra.util.statement_presentation.**available_sources_stmts**(*stmts*, *custom_sources=None*)
Returns the set of sources available in a list of statements

Return type Set[str]

indra.util.statement_presentation.**db_sources = ['psp', 'cbn', 'pc', 'bel_lc', 'signor', 'biogrid', 'tas', 'hprd', 'trrust', 'ctd', 'vhn', 'pe', 'drugbank', 'omnipath', 'conib', 'crog', 'dgi', 'minerva', 'creeds', 'ubibrowser', 'acsn']**
Database source names as they appear in the DB

indra.util.statement_presentation.**group_and_sort_statements**(*stmt_list*, *sort_by='default'*, *custom_stats=None*, *grouping_level='agent-pair'*)
Group statements by type and arguments, and sort by prevalence.

Parameters

- **stmt_list** (*list[Statement]*) – A list of INDRA statements.
- **sort_by** (*str or function or None*) – If str, it indicates which parameter to sort by, such as 'belief' or 'ev_count', or 'ag_count'. Those are the default options because they can be derived from a list of statements, however if you give a custom *stmt_metrics*, you may use any of the parameters used to build it. The default, 'default', is mostly a sort by ev_count but also favors statements with fewer agents. Alternatively, you may give a function that takes a dict as its single argument, a dictionary of metrics. These metrics are determined by the contents of the *stmt_metrics* passed as an argument (see StmtGroup for details), or else will contain the default metrics that can be derived from the statements themselves, namely *ev_count*, *belief*, and *ag_count*. The value may also be None, in which case the sort function will return the same value for all elements, and thus the original order of elements will be preserved. This could have strange effects when statements are grouped (i.e. when *grouping_level* is not 'statement'); such functionality is untested and we make no guarantee that it will work.
- **custom_stats** (*list[StmtStat]*) – A list of custom statement statistics to be used in addition to, or upon name conflict in place of, the default statement statistics derived from

the list of statements.

- **grouping_level** (*str*) – The options are 'agent-pair', 'relation', and 'statement'. These correspond to grouping by agent pairs, agent and type relationships, and a flat list of statements. The default is 'agent-pair'.

Returns **sorted_groups** – A list of tuples of the form (sort_param, key, contents, metrics), where the sort param is whatever value was calculated to sort the results, the key is the unique key for the agent pair, relation, or statements, and the contents are either relations, statements, or statement JSON, depending on the level. This structure is recursive, so the each list of relations will also follow this structure, all the way down to the lowest level (statement JSON). The metrics a dict of the aggregated metrics for the entry (e.g. source counts, evidence counts, etc).

Return type list[tuple]

indra.util.statement_presentation.**internal_source_mappings = {'bel': 'bel_lc', 'biopax': 'pc', 'phosphoelm': 'pe', 'phosphosite': 'psp', 'virhostnet': 'vhn'}**
Maps from source_info.json names to DB names

indra.util.statement_presentation.**make_standard_stats**(*ev_counts=None*, *beliefs=None*, *source_counts=None*)
Generate the standard ev_counts, beliefs, and source count stats.

indra.util.statement_presentation.**make_stmt_from_relation_key**(*relation_key*, *agents=None*)
Make a Statement from the relation key.

Specifically, make a Statement object from the sort key used by *group_and_sort_statements*.

indra.util.statement_presentation.**make_string_from_relation_key**(*rel_key*)
Make a Statement string via EnglishAssembler from the relation key.

Specifically, make a string from the key used by *group_and_sort_statements* for contents grouped at the relation level.

indra.util.statement_presentation.**make_top_level_label_from_names_key**(*names*)
Make an english string from the tuple names.

indra.util.statement_presentation.**reader_sources = ['geneways', 'tees', 'gnbr', 'isi', 'trips', 'rlimsp', 'medscan', 'eidos', 'sparser', 'reach']**
Reader source names as they appear in the DB

indra.util.statement_presentation.**reverse_source_mappings = {'bel_lc': 'bel', 'pc': 'biopax', 'pe': 'phosphoelm', 'psp': 'phosphosite', 'vhn': 'virhostnet'}**
Maps from db names to source_info.json names

indra.util.statement_presentation.**standardize_counts**(*counts*)
Standardize hash-based counts dicts to be int-keyed.

indra.util.statement_presentation.**stmt_to_english**(*stmt*)
Return an English assembled Statement as a sentence.

4.14.2 Utilities for using AWS (indra.util.aws)

class indra.util.aws.**JobLog**(*job_info*, *log_group_name='/aws/batch/job'*, *verbose=False*, *append_dumps=True*)

Gets the Cloudwatch log associated with the given job.

Parameters

- **job_info** (*dict*) – dict containing entries for 'jobName' and 'jobId', e.g., as returned by get_jobs()
- **log_group_name** (*string*) – Name of the log group; defaults to '/aws/batch/job'

Returns The event messages in the log, with the earliest events listed first.

Return type list of strings

dump(*out_file*, *append=None*)

Dump the logs in their entirety to the specified file.

load(*out_file*)

Load the log lines from the cached files.

indra.util.aws.**dump_logs**(*job_queue='run_reach_queue'*, *job_status='RUNNING'*)

Write logs for all jobs with given the status to files.

indra.util.aws.**get_batch_command**(*command_list*, *project=None*, *purpose=None*)

Get the command appropriate for running something on batch.

indra.util.aws.**get_date_from_str**(*date_str*)

Get a utc datetime object from a string of format %Y-%m-%d-%H-%M-%S

Parameters **date_str** (*str*) – A string of the format %Y(-%m-%d-%H-%M-%S). The string is assumed to represent a UTC time.

Return type datetime.datetime

indra.util.aws.**get_jobs**(*job_queue='run_reach_queue'*, *job_status='RUNNING'*)

Returns a list of dicts with jobName and jobId for each job with the given status.

indra.util.aws.**get_s3_client**(*unsigned=True*)

Return a boto3 S3 client with optional unsigned config.

Parameters **unsigned** (*Optional[bool]*) – If True, the client will be using unsigned mode in which public resources can be accessed without credentials. Default: True

Returns A client object to AWS S3.

Return type botocore.client.S3

indra.util.aws.**get_s3_file_tree**(*s3*, *bucket*, *prefix*, *date_cutoff=None*, *after=True*, *with_dt=False*)

Overcome s3 response limit and return NestedDict tree of paths.

The NestedDict object also allows the user to search by the ends of a path.

The tree mimics a file directory structure, with the leave nodes being the full unbroken key. For example, 'path/to/file.txt' would be retrieved by

ret['path']['to']['file.txt']['key']

The NestedDict object returned also has the capability to get paths that lead to a certain value. So if you wanted all paths that lead to something called 'file.txt', you could use

ret.get_paths('file.txt')

For more details, see the NestedDict docs.

Parameters

- **s3** (*boto3.client.S3*) – A boto3.client.S3 instance
- **bucket** (*str*) – The name of the bucket to list objects in
- **prefix** (*str*) – The prefix filtering of the objects for list
- **date_cutoff** (*str|datetime.datetime*) – A datestring of format %Y(-%m-%d-%H-%M-%S) or a datetime.datetime object. The date is assumed to be in UTC. By default no filtering is done. Default: None.
- **after** (*bool*) – If True, only return objects after the given date cutoff. Otherwise, return objects before. Default: True
- **with_dt** (*bool*) – If True, yield a tuple (key, datetime.datetime(LastModified)) of the s3 Key and the object's LastModified date as a datetime.datetime object, only yield s3 key otherwise. Default: False.

Returns A file tree represented as an NestedDict

Return type *NestedDict*

indra.util.aws.**iter_s3_keys**(*s3*, *bucket*, *prefix*, *date_cutoff=None*, *after=True*, *with_dt=False*, *do_retry=True*)

Iterate over the keys in an s3 bucket given a prefix

Parameters

- **s3** (*boto3.client.S3*) – A boto3.client.S3 instance
- **bucket** (*str*) – The name of the bucket to list objects in
- **prefix** (*str*) – The prefix filtering of the objects for list
- **date_cutoff** (*str|datetime.datetime*) – A datestring of format %Y(-%m-%d-%H-%M-%S) or a datetime.datetime object. The date is assumed to be in UTC. By default no filtering is done. Default: None.
- **after** (*bool*) – If True, only return objects after the given date cutoff. Otherwise, return objects before. Default: True
- **with_dt** (*bool*) – If True, yield a tuple (key, datetime.datetime(LastModified)) of the s3 Key and the object's LastModified date as a datetime.datetime object, only yield s3 key otherwise. Default: False.
- **do_retry** (*bool*) – If True, and no contents appear, try again in case there was simply a brief lag. If False, do not retry, and just accept the "directory" is empty.

Returns An iterator over s3 keys or (key, LastModified) tuples.

Return type iterator[key]|iterator[(key, datetime.datetime)]

indra.util.aws.**kill_all**(*job_queue*, *reason='None given'*, *states=None*, *kill_list=None*)

Terminates/cancels all jobs on the specified queue.

Parameters

- **job_queue** (*str*) – The name of the Batch job queue on which you wish to terminate/cancel jobs.
- **reason** (*str*) – Provide a reason for the kill that will be recorded with the job's record on AWS.

- **states** (*None or list[str]*) – A list of job states to remove. Possible states are 'STARTING', 'RUNNABLE', and 'RUNNING'. If None, all jobs in all states will be ended (modulo the *kill_list* below).
- **kill_list** (*None or list[dict]*) – A list of job dictionaries (as returned by the submit function) that you specifically wish to kill. All other jobs on the queue will be ignored. If None, all jobs on the queue will be ended (modulo the above).

Returns **killed_ids** – A list of the job ids for jobs that were killed.

Return type list[str]

indra.util.aws.**rename_s3_prefix**(*s3*, *bucket*, *old_prefix*, *new_prefix*)
Change an s3 prefix within the same bucket.

indra.util.aws.**tag_instance**(*instance_id*, ***tags*)
Tag a single ec2 instance.

indra.util.aws.**tag_myself**(*project='aske'*, ***other_tags*)
Function run when indra is used in an EC2 instance to apply tags.

4.14.3 A utility to get the INDRA version (indra.util.get_version)

This tool provides a uniform method for createing a robust indra version string, both from within python and from commandline. If possible, the version will include the git commit hash. Otherwise, the version will be marked with 'UNHASHED'.

indra.util.get_version.**get_git_info**()
Get a dict with useful git info.

indra.util.get_version.**get_version**(*with_git_hash=True*, *refresh_hash=False*)
Get an indra version string, including a git hash.

4.14.4 Define NestedDict (indra.util.nested_dict)

class indra.util.nested_dict.**NestedDict**
A dict-like object that recursively populates elements of a dict.

More specifically, this acts like a recursive defaultdict, allowing, for example:

```
>>> nd = NestedDict()
>>> nd['a']['b']['c'] = 'foo'
```

In addition, useful methods have been defined that allow the user to search the data structure. Note that the are not particularly optimized methods at this time. However, for convenience, you can for example simply call *get_path* to get the path to a particular key:

```
>>> nd.get_path('c')
(('a', 'b', 'c'), 'foo')
```

and the value at that key. Similarly:

```
>>> nd.get_path('b')
(('a', 'b'), NestedDict(
  'c': 'foo'
))
```

get, *gets*, and *get_paths* operate on similar principles, and are documented below.

export_dict()
Convert this into an ordinary dict (of dicts).

get(*key*)
Find the first value within the tree which has the key.

get_leaves()
Get the deepest entries as a flat set.

get_path(*key*)
Like *get*, but also return the path taken to the value.

get_paths(*key*)
Like *gets*, but include the paths, like *get_path* for all matches.

gets(*key*)
Like *get*, but return all matches, not just the first.

4.14.5 Shorthands for plot formatting (indra.util.plot_formatting)

indra.util.plot_formatting.**format_axis**(*ax*, *label_padding=2*, *tick_padding=0*, *yticks_position='left'*)
Set standardized axis formatting for figure.

indra.util.plot_formatting.**set_fig_params**()
Set standardized font properties for figure.

CHAPTER

FIVE

TUTORIALS

5.1 Using natural language to build models

In this tutorial we build a simple model using natural language, and export it into different formats.

5.1.1 Read INDRA Statements from a natural language string

First we import INDRA's API to the TRIPS reading system. We then define a block of text which serves as the description of the mechanism to be modeled in the *model_text* variable. Finally, *indra.sources.trips.process_text* is called which sends a request to the TRIPS web service, gets a response and processes the extraction knowledge base to obtain a list of INDRA Statements

```
In [1]: from indra.sources import trips

In [2]: model_text = 'MAP2K1 phosphorylates MAPK1 and DUSP6 dephosphorylates MAPK1.'

In [3]: tp = trips.process_text(model_text)
```

At this point *tp.statements* should contain 2 INDRA Statements: a Phosphorylation Statement and a Dephosphorylation Statement. Note that the evidence sentence for each Statement is propagated:

```
In [4]: for st in tp.statements:
   ...:     print('%s with evidence "%s"' % (st, st.evidence[0].text))
   ...:
Phosphorylation(MAP2K1(), MAPK1()) with evidence "MAP2K1 phosphorylates MAPK1 and DUSP6␣
↪dephosphorylates MAPK1."
Dephosphorylation(DUSP6(), MAPK1()) with evidence "MAP2K1 phosphorylates MAPK1 and DUSP6␣
↪dephosphorylates MAPK1."
```

5.1.2 Assemble the INDRA Statements into a rule-based executable model

We next use INDRA's PySB Assembler to automatically assemble a rule-based model representing the biochemical mechanisms described in *model_text*. First a PysbAssembler object is instantiated, then the list of INDRA Statements is added to the assembler. Finally, the assembler's *make_model* method is called which assembles the model and returns it, while also storing it in *pa.model*. Notice that we are using *policies='two_step'* as an argument of *make_model*. This directs the assemble to use rules in which enzymatic catalysis is modeled as a two-step process in which enzyme and substrate first reversibly bind and the enzyme-substrate complex produces and releases a product irreversibly.

```
In [5]: from indra.assemblers.pysb import PysbAssembler

In [6]: pa = PysbAssembler()

In [7]: pa.add_statements(tp.statements)

In [8]: pa.make_model(policies='two_step')
Out[8]: <Model 'indra_model' (monomers: 3, rules: 6, parameters: 9, expressions: 0,␣
↪compartments: 0) at 0x7f53e12d41d0>
```

At this point *pa.model* contains a PySB model object with 3 monomers,

```
In [9]: for monomer in pa.model.monomers:
   ...:     print(monomer)
   ...:
Monomer('MAP2K1', ['mapk'])
Monomer('MAPK1', ['phospho', 'map2k', 'dusp'], {'phospho': ['u', 'p']})
Monomer('DUSP6', ['mapk'])
```

6 rules,

```
In [10]: for rule in pa.model.rules:
   ....:     print(rule)
   ....:
Rule('MAP2K1_phosphorylation_bind_MAPK1_phospho', MAP2K1(mapk=None) + MAPK1(phospho='u',␣
↪map2k=None) >> MAP2K1(mapk=1) % MAPK1(phospho='u', map2k=1), kf_mm_bind_1)
Rule('MAP2K1_phosphorylation_MAPK1_phospho', MAP2K1(mapk=1) % MAPK1(phospho='u',␣
↪map2k=1) >> MAP2K1(mapk=None) + MAPK1(phospho='p', map2k=None), kc_mm_phosphorylation_
↪1)
Rule('MAP2K1_dissoc_MAPK1', MAP2K1(mapk=1) % MAPK1(map2k=1) >> MAP2K1(mapk=None) +␣
↪MAPK1(map2k=None), kr_mm_bind_1)
Rule('DUSP6_dephosphorylation_bind_MAPK1_phospho', DUSP6(mapk=None) + MAPK1(phospho='p',␣
↪dusp=None) >> DUSP6(mapk=1) % MAPK1(phospho='p', dusp=1), kf_dm_bind_1)
Rule('DUSP6_dephosphorylation_MAPK1_phospho', DUSP6(mapk=1) % MAPK1(phospho='p', dusp=1)␣
↪>> DUSP6(mapk=None) + MAPK1(phospho='u', dusp=None), kc_dm_phosphorylation_1)
Rule('DUSP6_dissoc_MAPK1', DUSP6(mapk=1) % MAPK1(dusp=1) >> DUSP6(mapk=None) +␣
↪MAPK1(dusp=None), kr_dm_bind_1)
```

and 9 parameters (6 kinetic rate constants and 3 total protein amounts) that are set to nominal but plausible values,

```
In [11]: for parameter in pa.model.parameters:
   ....:     print(parameter)
   ....:
Parameter('kf_mm_bind_1', 1e-06)
Parameter('kr_mm_bind_1', 0.1)
Parameter('kc_mm_phosphorylation_1', 100.0)
Parameter('kf_dm_bind_1', 1e-06)
Parameter('kr_dm_bind_1', 0.1)
Parameter('kc_dm_phosphorylation_1', 100.0)
Parameter('MAP2K1_0', 10000.0)
Parameter('MAPK1_0', 10000.0)
Parameter('DUSP6_0', 10000.0)
```

The model also contains extensive annotations that tie the monomers to database identifiers and also annotate the

semantics of each component of each rule.

```
In [12]: for annotation in pa.model.annotations:
   ....:     print(annotation)
   ....:
Annotation(MAP2K1, 'https://identifiers.org/hgnc:6840', 'is')
Annotation(MAP2K1, 'https://identifiers.org/uniprot:Q02750', 'is')
Annotation(MAP2K1, 'https://identifiers.org/ncit:C17808', 'is')
Annotation(MAPK1, 'https://identifiers.org/hgnc:6871', 'is')
Annotation(MAPK1, 'https://identifiers.org/uniprot:P28482', 'is')
Annotation(MAPK1, 'https://identifiers.org/ncit:C17589', 'is')
Annotation(DUSP6, 'https://identifiers.org/hgnc:3072', 'is')
Annotation(DUSP6, 'https://identifiers.org/uniprot:Q16828', 'is')
Annotation(DUSP6, 'https://identifiers.org/ncit:C106024', 'is')
Annotation(MAP2K1_phosphorylation_bind_MAPK1_phospho, 'eecbf584-4db7-416c-b24a-
↪dceafdd0fd76', 'from_indra_statement')
Annotation(MAP2K1_phosphorylation_MAPK1_phospho, 'MAP2K1', 'rule_has_subject')
Annotation(MAP2K1_phosphorylation_MAPK1_phospho, 'MAPK1', 'rule_has_object')
Annotation(MAP2K1_phosphorylation_MAPK1_phospho, 'eecbf584-4db7-416c-b24a-dceafdd0fd76',
↪'from_indra_statement')
Annotation(MAP2K1_dissoc_MAPK1, 'eecbf584-4db7-416c-b24a-dceafdd0fd76', 'from_indra_
↪statement')
Annotation(DUSP6_dephosphorylation_bind_MAPK1_phospho, '858e1630-3fe4-4faa-8ead-
↪4cbb26d08b24', 'from_indra_statement')
Annotation(DUSP6_dephosphorylation_MAPK1_phospho, 'DUSP6', 'rule_has_subject')
Annotation(DUSP6_dephosphorylation_MAPK1_phospho, 'MAPK1', 'rule_has_object')
Annotation(DUSP6_dephosphorylation_MAPK1_phospho, '858e1630-3fe4-4faa-8ead-4cbb26d08b24',
↪ 'from_indra_statement')
Annotation(DUSP6_dissoc_MAPK1, '858e1630-3fe4-4faa-8ead-4cbb26d08b24', 'from_indra_
↪statement')
```

5.1.3 Exporting the model into other common formats

From the assembled PySB format it is possible to export the model into other common formats such as SBML, BNGL and Kappa. One can also generate a Matlab or Mathematica script with ODEs corresponding to the model.

```
pa.export_model('sbml')
pa.export_model('bngl')
```

One can also pass a file name argument to the *export_model* function to save the exported model directly into a file:

```
pa.export_model('sbml', 'example_model.sbml')
```

5.2 The Statement curation interface

You will usually access this interface from an INDRA application that exposes statements to you. However if you just want to try out the interface or don't want to take the detour through any of the applications, you can follow the format below to access the interface directly in your browser from the INDRA-DB REST API:

```
http://api.host/statements/from_agents?subject=SUBJ&object=OBJ&api_key=12345&format=html
```

where *api.host* should be replaced with the address to the REST API service (see the documentation). Entering the whole address in your browser will query for statements where *SUBJ* is the subject and *OBJ* is the object of the statements.

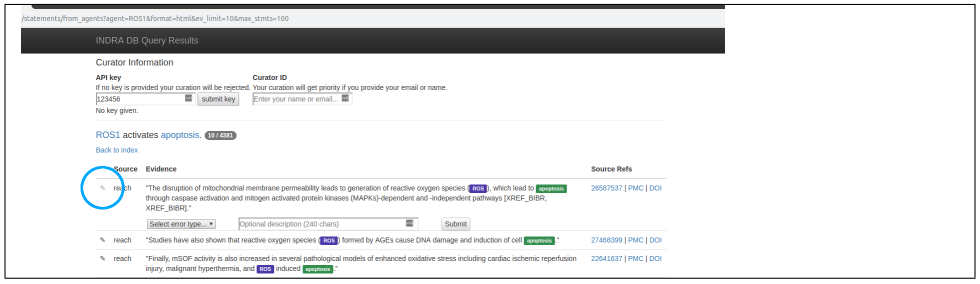
For more details about what options are available when doing curation, please refer to the curation section of the documentation.

5.2.1 Curating a Statement

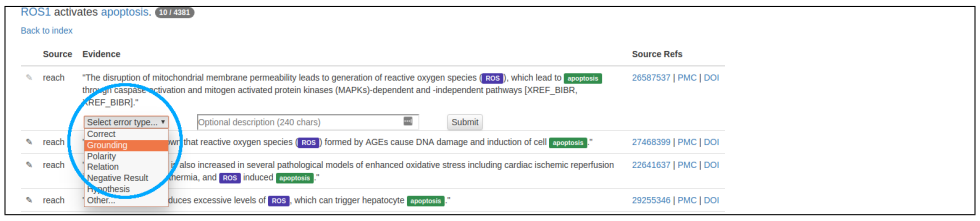
Let's assume you want to check any statements were ROS1 is an agent for errors. Let's also limit the number of statements to 100 and the number of evidences per statements to 5. This will speed up the query and page loading. The appropriate address to enter in your browser would then be:

```
http://api.host/statements/from_agents?agent=ROS1&format=html&ev_limit=5&max_stmts=100
```

To start curating a statement, **click the pen icon (circled)** on the far left side of the statement. This will produce a row below the statement with a dropdown menu, a text box and a submit button:



The **dropdown menu** contains common errors and also the possibility to mark the statement as 'correct'. If none of the types fit, select the *other...* option, and describe the error with one or a few words in the provided textbox. Note that if you pick *other...*, describing the error is mandatory. In our example, we see that *reactive oxygen species* is incorrectly grounded to *ROS*, so we pick *grounding* from the dropdown menu:



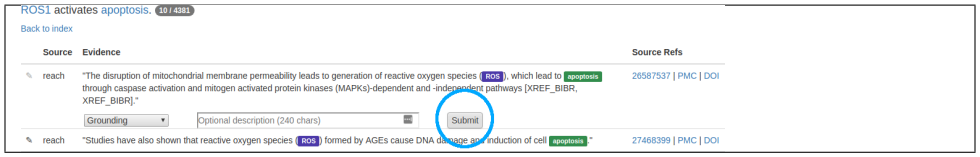
In the textbox, you can add a short optional description to clarify why you marked this piece of evidence with the error type you chose. When you are done, you are ready to submit your curation.

5.2.2 Submitting a Curation

To **submit a curation**, you will need to at least make a **selection in the dropdown menu** (by the curated statement). You will also need to be logged in before the curation is submitted. If you do not already have an account, all we ask for is your email.

If you selected *other...* in the dropdown menu, you must *also* describe the error in the textbox.

When you have entered the necessary information, click the **Submit button** by the statement that you curated (if you aren't logged in, you will be prompted to do so at this point):



A status message will appear once the server has processed the submission, indicating if the submission was successful or which problem arose if not. The pen icon will also change color based in the returned status. **Green** indicates a successful submission:

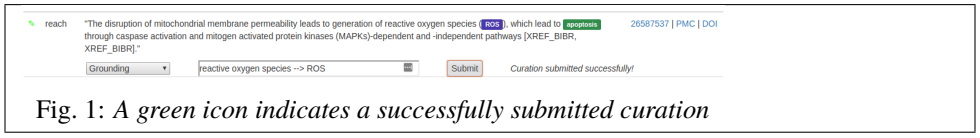


Fig. 1: *A green icon indicates a successfully submitted curation*

while a **red** indicates something went wrong with the submission:

Fig. 2: *A red icon indicates that something went wrong during the submission*

5.2.3 Curation Guidelines

Basic principles

The main question to ask when deciding whether a given Statement is correct with respect to a given piece of evidence is:

```
Is there support in the evidence sentence for the Statement?
```

If the answer is **Yes**, then the given sentence is a valid piece of evidence for the Statement. In fact, you can assert this correctness by choosing the "Correct" option from the curation drop-down list. Curations that assert correctness are just as valuable as curations of incorrectness so the use of this option is encouraged.

Assuming the answer to the above question is **No**, one needs to determine what the error can be attributed to. The following section describes the specific error types that can be flagged.

Types of errors to curate

There are currently the following options to choose from when curating incorrect Statement-sentence relationships:

- **Entity Boundaries**: this is applicable if the bounderies of one of the named entities was incorrectly recognized. Example: "gap" is highlighted as an entity, when in fact, the entity mentioned in the sentence was "gap junction". These errors in entity boundaries almost always result in incorrect grounding, since the wrong string is attempted to be grounded. Therefore this error "subsumes" grounding errors. Note: to help correct entity boundaries, add the following to the Optional description text box: [gap junction], i.e. the desired entity name inside square brackets.
- **Grounding**: this is applicable if a named entity is assigned an incorrect database identifier. Example:

```
Assume that in a sentence, "ER" is mentioned referring to endoplasmic
reticulum, but in a Statement extracted from the sentence, it is
grounded to the ESR1 (estrogen receptor alpha) gene.
```

  Note: to help correct grounding, add the following to the Optional description text box:

```
[ER] -> MESH:D004721
```

  where [ER] is the entity string, MESH is the namespace of a database/ontology, and D004721 is the unique ID corresponding to endoplasmic reticulum in MESH. A list of commonly used namespaces in INDRA are given in: https://indra.readthedocs.io/en/latest/modules/statements.html. Note that you can also add multiple groundings separated by "|", e.g. HGNC:11998|UP:P04637.

- **Polarity**: this is applicable if an essentially correct Statement was extracted but the Statement has the wrong polarity, e.g. Activation instead of Inhibition, of Phosphorylation instead of Dephosphorylation. Example:

```
Sentence: "NDRG2 overexpression specifically inhibits SOCS1 phosphorylation"
Statement: Phosphorylation(NDRG2(), SOCS1())
```

  has incorrect polarity. It should be Dephosphorylation instead of Phosphorylation.

- **No Relation**: this is applicable if the sentence does not imply a relationship between the agents appearing in the Statement. Example:

```
Sentence: "Furthermore, triptolide mediated inhibition of NF-kappaB
    activation, Stat3 phosphorylation and increase of SOCS1 expression in
    DC may be involved in the inhibitory effect of triptolide."
Statement: Phosphorylation(STAT3(), SOCS1())
```

  can be flagged as No Relation.

- **Wrong Relation Type**: this is applicable if the sentence implies a relationship between agents appearing in the Statement but the type of Statement is inconsistent with the sentence. Example:

```
Sentence: "We report the interaction between tacrolimus and chloramphenicol
    in a renal transplant recipient."
Statement: Complex(tacrolimus(), chloramphenicol())
```

  can be flagged as Wrong Relation Type since the sentence implies a drug interaction that does not involve complex formation.

- **Activity vs. Amount**: this is applicable when the sentence implies a regulation of amount but the corresponding Statement implies regulation of activity or vice versa. Example:

```
Sentence: "NFAT upregulates IL2"
Sentence: Activation(NFAT(), IL2())
```

Here the sentence implies upregulation of the amount of IL2 but the corresponding Statement is of type Activation rather than IncreaseAmount.

- **Negative Result**: this is applicable if the sentence implies the lack of or opposite of a relationship. Example:

```
Sentence: "These results indicate that CRAF, but not BRAF phosphorylates
    MEK in NRAS mutated cells."
Statement: Phosphorylation(BRAF(), MEK())
```

Here the sentence does not support the Statement due to a negation and should therefore be flagged as a Negative Result.

- **Hypothesis**: this is applicable if the sentence describes a hypothesis or an experiment rather than a result or mechanism. Example:

```
Sentence: "We tested whether EGFR activates ERK."
Statement: Activation(EGFR(), ERK())
```

Here the sentence describes a hypothesis with respect to the Statement, and should therefore be flagged as a Hypothesis upon curation (unless of course the Statement already has a correct *hypothesis* flag).

- **Agent Conditions**: this is applicable if one of the Agents in the Statement is missing relevant conditions that are mentioned in the sentence, or has incorrect conditions attached to it. Example:

```
Sentence: "Mutant BRAF activates MEK"
Statement: Activation(BRAF(), MEK())
```

can be curated to be missing Agent conditions since the mutation on BRAF is not captured.

- **Modification Site**: this is applicable if an amino-acid site is missing or incorrect in a modification Statement. Example:

```
Sentence: "MAP2K1 phosphorylates MAPK1 at T185."
Statement: Phosphorylation(MAP2K1(), MAPK1())
```

Here the obvious modification site is missing from MAPK1.

- **Other**: this is an option you can choose whenever the problem isn't well captured by any of the more specific options. In this case you need to add a note to explain what the issue is.

General notes on curation

- If you spot multiple levels of errors in a Statement-sentence pair, use the most relevant error type in the dropdown menu. E.g. if you see both a grounding error and a polarity error, you should pick the grounding error since a statement with a grounding error generally would not exist if the grounding was correct.
- If you still feel like multiple errors are appropriate for the curation, select a new error from the dropdown menu and make a new submission.
- Please be consistent in using your email address as your curator ID. Keeping track of who curated what helps us to faster track down issues with readers and the assembly processes that generate statements.

5.3 Assembling everything known about a particular gene

Assume you are interested in collecting all mechanisms that a particular gene is involved in. Using INDRA, it is possible to collect everything curated about the gene in pathway databases and then read all the accessible literature discussing the gene of interest. This knowledge is aggregated as a set of INDRA Statements which can then be assembled into several different model and network formats and possibly shared online.

For the sake of this example, assume that the gene of interest is H2AX.

It is important to use the standard HGNC gene symbol of the gene throughout the example (this information is available on http://www.genenames.org/ or http://www.uniprot.org/) - abritrary synonyms will not work!

5.3.1 Collect mechanisms from PathwayCommons and the BEL Large Corpus

We first collect Statements from the PathwayCommons database via INDRA's BioPAX API and then collect Statements from the BEL Large Corpus via INDRA's BEL API.

```
from indra.tools.gene_network import GeneNetwork

gn = GeneNetwork(['H2AX'])
biopax_stmts = gn.get_biopax_stmts()
bel_stmts = gn.get_bel_stmts()
```

at this point *biopax_stmts* and *bel_stmts* are two lists of INDRA Statements.

5.3.2 Collect a list of publications that discuss the gene of interest

We next use INDRA's literature client to find PubMed IDs (PMIDs) that discuss the gene of interest. To find articles that are annotated with the given gene, INDRA first looks up the Entrez ID corresponding to the gene name and then finds associated publications.

```
from indra import literature

pmids = literature.pubmed_client.get_ids_for_gene('H2AX')
```

The variable *pmids* now contains a list of PMIDs associated with the gene.

5.3.3 Get the abstracts corresponding to the publications

Next we use INDRA's literature client to fetch the abstracts corresponding to the PMIDs we have just collected. The client also returns other content types, like xml, for full text (if available). Here we cut the list of PMIDs short to just the first 10 IDs that contain abstracts to make the processing faster.

```
from indra import literature

paper_contents = {}
for pmid in pmids:
    content, content_type = literature.get_full_text(pmid, 'pmid')
    if content_type == 'abstract':
        paper_contents[pmid] = content
    if len(paper_contents) == 10:
        break
```

We now have a dictionary called *paper_contents* which stores the content for each PMID we looked up. While the abstracts are in plain text format, some content is sometimes returned in different either PMC NXML or Elsevier XML format. To process XML from different sources, some example are: INDRA Reach API or the INDRA Elsevier client.

5.3.4 Read the content of the publications

We next run the REACH reading system on the publications. Here we assume that the REACH web service is running locally and is available at *http://localhost:8080* (the default web service endpoints for processing text and nxml are available as importable variables e.g., *local_text_url*. To get started wtih this, see method 1 listed in <INDRA Reach API documentation.

```
from indra.sources import reach

literature_stmts = []
for pmid, content in paper_contents.items():
    rp = reach.process_text(content, url=reach.local_text_url)
    literature_stmts += rp.statements
print('Got %d statements' % len(literature_stmts))
```

The list *literature_stmts* now contains the results of all the statements that were read.

5.3.5 Combine all statements and run pre-assembly

```
from indra.tools import assemble_corpus as ac

stmts = biopax_stmts + bel_stmts + literature_stmts

stmts = ac.map_grounding(stmts)
stmts = ac.map_sequence(stmts)
stmts = ac.run_preassembly(stmts)
```

At this point *stmts* contains a list of Statements with grounding, having been mapped according to INDRA's built in grounding map and disambiguation features, amino acid sites having been mapped, duplicates combined, and hierarchically subsumed variants of statements hidden. It is possible to run other assembly steps and filters on the results such as to keep only human genes, remove Statements with ungrounded genes, or to keep only certain types of interactions. You can find more assembly steps that can be included in your pipeline in the Assemble Corpus documentation. You can also read more about the pre-assembly process in the preassembly module documentation and in the GitHub documentation

5.3.6 Assemble the statements into a network model

CX Network Model

We can assemble the statements into e.g., a CX network model:

```
from indra.assemblers.cx import CxAssembler
from indra.databases import ndex_client

cxa = CxAssembler(stmts)
cx_str = cxa.make_model()
```

We can now upload this network to the Network Data Exchange (NDEx).

```
ndex_cred = {'user': 'myusername', 'password': 'xxx'}
network_id = ndex_client.create_network(cx_str, ndex_cred)
print(network_id)
```

IndraNet Model

Another network model that can assembled is the IndraNet graph which is a light-weight networkx derived object.

```
from indra.assemblers.indranet import IndraNetAssembler
indranet_assembler = IndraNetAssembler(statements=stmts)
indranet = indranet_assembler.make_model()
```

Since the IndraNet class is a child class of a networkx Graph, one can use networkx's algorithms:

```
import networkx as nx
paths = nx.single_source_shortest_path(G=indranet, source='H2AX',
                                       cutoff=1)
```

Executable PySB Model

An executable PySB model can be assembled with the PySB assembler:

```
from indra.assemblers.pysb import PysbAssembler
pysb = PysbAssembler(statements=stmts)
pysb_model = pysb.make_model()
```

Read more about PySB models in the PySB documentation and look into the natural language modeling tutorial which uses PySB models.

Read more about all assembly output formats in the README and in the module references.

CHAPTER

SIX

REST API

Many functionalities of INDRA can be used via a REST API. This enables making use of INDRA's knowledge sources and assembly capabilities in a RESTful, platform independent fashion. The REST service is available as a public web service at http://api.indra.bio:8000 and can also be run locally.

6.1 Local installation and use

Running the REST service requires the *flask*, *flask_restx*, *flask_cors* and *docstring-parser* packages to be installed in addition to all the other requirements of INDRA. The REST service can be launched by running *api.py* in the *rest_api* folder within *indra*.

As an alternative, the REST service can be run via the INDRA Docker without the need for installing any dependencies as follows:

```
docker pull labsyspharm/indra
docker run -id -p 8080:8080 --entrypoint python labsyspharm/indra /sw/indra/rest_api/api.
↪py
```

6.2 Documentation

The specific end-points and input/output parameters offered by the REST API are documented at http://api.indra.bio:8000 or the local address on which the API is running.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[wang2016] Wang, Z., *et al*. (2016). Extraction and analysis of signatures from the Gene Expression Omnibus by the crowd. *Nature Communications*, **7**(1), 12846.

PYTHON MODULE INDEX

i

INDEX

A

B

C

D

F

G

H

I

J

K

L

M

N

O

P

S

T

U

V

W