INDRA Documentation

Release 1.10.0

B. M. Gyori, J. A. Bachman

Dec 14, 2018

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 1

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 2

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 both Python 2 and 3 (tested with 2.7 and 3.5).

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 .
```

Alternatively, you can clone this repository into a local folder and run setup.py from the terminal as

```
git clone https://github.com/sorgerlab/indra.git
cd indra
python setup.py install
```

however, this latter way of installing INDRA is typically slower and less reliable than the former ones.

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 either setup method (running setup.py install or using 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

To be able to use INDRA's BioPAX API and optional offline reading via the REACH and Eidos APIs, an additional package called pyjnius is needed to allow using Java/Scala classes from Python. This is only strictly required in these input sources and the rest of INDRA will work without pyjnius.

1. Install JRE and JDK 8 from Oracle. Pyjnius is currently incompatible with Java 9, so make sure to get Java 8.

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

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_74.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>
...
```

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

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home
```

4. Then first install Cython (tested with version 0.28.1) followed by pyjnius (note that the released version of pyjnius does _not_ work with recent Cython versions, hence installation from Github is required). These need to be broken up into two sequential calls to pip install.

```
pip install cython
pip install pjnius==1.1.4
```

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 Matplotlib

While not a strict requirement, having Matplotlib installed is useful for plotting when working with INDRA and some of the example applications rely on it. It can be installed as

```
pip install matplotlib
```

2.3.5 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
biopax	BioPAX input processing and Pathway Commons queries
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
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 3

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 PMC3717945.

```
from indra.sources import reach
reach_processor = reach.process_pmc('3717945')
```

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 Getting paths between two proteins from PathwayCommons (BioPAX)

In this example, we search for paths between the BRAF and MAPK3 proteins in the PathwayCommons databases using INDRA's BioPAX API. Note that this example will only work if all dependencies of the indra.sources.biopax module are installed.

See the Installation instructions for more details.

```
from indra.sources import biopax
proteins = ['BRAF', 'MAPK3']
limit = 2
biopax_processor = biopax.process_pc_pathsbetween(proteins, limit)
```

We passed the second argument *limit = 2*, which defines the upper limit on the length of the paths that are searched. By default the limit is 1. The *biopax_processor* object has a *statements* attribute which contains a list of INDRA Statements extracted from the queried paths.

3.2.5 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)
```

3.2.6 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.

CHAPTER 4

INDRA modules reference

4.1 INDRA Statements (`indra.statements`)

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:

- *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*. 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*.

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

Type	Database	Example
Gene/Protein	HGNC	{'HGNC': '11998'}
Gene/Protein	UniProt	{'UP': 'P04637'}
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 / HMS-LINCS	{'LINCS': '42611257'}
Metabolite	HMDB	{'HMDB': 'HMDB00122'}
Process, location, etc.	GO	{'GO': 'GO:0006915'}
Process, disease, etc.	MeSH	{'MESH': 'D008113'}
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.

4.1.1 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.**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.**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.**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.**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.**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 canonicallized agent list.

get_hash(*shallow=True*, *refresh=False*)
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.

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, belief, 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.

to_graph()

Return Statement as a networkx graph.

to_json(*use_sbo=False*)

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

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

Return type dict

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

Bases: *indra.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*)
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

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

Return type dict

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

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

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

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

Return type dict

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

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

Bases: *indra.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.**Dephosphorylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.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.**Hydroxylation**(*enz*, *sub*, *residue=None*, *position=None*, *evidence=None*)

Bases: *indra.statements.AddModification*

Hydroxylation modification.

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

Bases: *indra.statements.RemoveModification*

Dehydroxylation modification.

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

Bases: *indra.statements.AddModification*

Sumoylation modification.

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

Bases: *indra.statements.RemoveModification*

Desumoylation modification.

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

Acetylation modification.

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

Deacetylation modification.

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

Glycosylation modification.

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

Deglycosylation modification.

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

Ribosylation modification.

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

Deribosylation modification.

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

Ubiquitination modification.

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

Deubiquitination modification.

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

Farnesylation modification.

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

Defarnesylation modification.

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

Geranylgeranylation modification.

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

Degeranylgeranylation modification.

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

Palmitoylation modification.

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

Depalmitoylation modification.

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

Myristoylation modification.

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

Demyristoylation modification.

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

Methylation modification.

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

Demethylation modification.

class indra.statements.**RegulateActivity**
Bases: *indra.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*)
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

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

Return type dict

class indra.statements.**Inhibition**(*subj*, *obj*, *obj_activity='activity'*, *evidence=None*)
Bases: *indra.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** (list of *Evidence*) – Evidence objects in support of the modification.

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

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

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

Return type dict

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

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

Return type dict

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

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

Return type dict

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

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

Return type dict

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

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

Return type dict

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

Superclass handling operations on directed, two-element interactions.

to_json(*use_sbo=False*)
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

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

Return type dict

class indra.statements.**DecreaseAmount**(*subj*, *obj*, *evidence=None*)
Bases: *indra.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** (list of *Evidence*) – Evidence objects in support of the degradation statement.

class indra.statements.**IncreaseAmount**(*subj*, *obj*, *evidence=None*)
Bases: *indra.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** (list of *Evidence*) – Evidence objects in support of the synthesis statement.

class indra.statements.**Influence**(*subj*, *obj*, *subj_delta=None*, *obj_delta=None*, *evidence=None*)
Bases: *indra.statements.IncreaseAmount*

An influence on the quantity of a concept of interest.

Parameters

- **subj** (indra.statement.Concept) – The concept which acts as the influencer.
- **obj** (indra.statement.Concept) – The concept which acts as the influencee

- **subj_delta** (*Optional[dict]*) – A dictionary specifying the polarity and magnitude of change in the subject.
- **obj_delta** (*Optional[dict]*) – A dictionary specifying the polarity and magnitude of change in the object.
- **evidence** (list of *Evidence*) – Evidence objects in support of the statement.

to_json(*use_sbo=False*)
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

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

Return type dict

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

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

Return type dict

class indra.statements.**Unresolved**(*uuid_str=None*, *shallow_hash=None*, *full_hash=None*)
Bases: *indra.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 place-holder, carrying only the uuid of the statement.

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

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

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

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

Invalid cellular component name.

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

Invalid residue (amino acid) name.

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

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

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.**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.
- **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.**BioContext**(*location=None*, *cell_line=None*, *cell_type=None*, *organ=None*, *disease=None*, *species=None*)
Bases: *indra.statements.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.**WorldContext**(*time=None*, *geo_location=None*)
Bases: *indra.statements.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

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

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**
Bases: object

An abstract class for Contexts.

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

indra.statements.**stmts_to_json**(*stmts_in*, *use_sbo=False*)
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

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

Return type dict

indra.statements.**stmts_from_json_file**(*fname*)
Return a list of statements loaded from a JSON file.

Parameters **fname** (*str*) – Path to the JSON file to load statements from.

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

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

indra.statements.**stmts_to_json_file**(*stmts*, *fname*)
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** (*str*) – Path to the JSON file to serialize Statements into.

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

indra.statements.**get_valid_location**(*location*)
Check if the given location represents a valid cellular component.

indra.statements.**get_valid_location**(*location*)
Check if the given location represents a valid cellular component.

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

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

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

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 Biology-oriented Reading Systems

REACH (indra.sources.reach)

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*)

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

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*)

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

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*, *output_fname='reach_output.json'*)

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 ran offline. Otherwise (by default) the web service is called. Default: False
- **output_fname** (*Optional[str]*) – The file to output the REACH JSON output to. Defaults to reach_output.json in current working directory.

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*, *output_fname='reach_output.json'*)

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 ran offline. Otherwise (by default) the web service is called. Default: False
- **output_fname** (*Optional[str]*) – The file to output the REACH JSON output to. Defaults to reach_output.json in current working directory.

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*, *output_fname='reach_output.json'*)

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: 3717945, PMC3717945 https://www.ncbi.nlm.nih.gov/pmc/
- **offline** (*Optional[bool]*) – If set to True, the REACH system is ran offline. Otherwise (by default) the web service is called. Default: False

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*, *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 ran offline. Otherwise (by default) the web service is called. Default: False
- **output_fname** (*Optional[str]*) – The file to output the REACH JSON output to. Defaults to reach_output.json in current working directory.

- ****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*, *output_fname='reach_output.json'*, *timeout=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 ran offline. Otherwise (by default) the web service is called. Default: False
- **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.

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*)

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
objectpath.Tree – The objectpath Tree object representing the extractions.

statements
list[indra.statements.Statement] – A list of INDRA Statements that were extracted by the processor.

citation
str – The PubMed ID associated with the extractions.

all_events
dict[str, str] – The frame IDs of all events by type in the REACH extraction.

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*)

position
Alias for field number 1

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

REACH reader (indra.sources.reach.reader)

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
org.clulab.reach.apis.ApiRuler – An instance of the REACH ApiRuler class (java object).

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'*)

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.

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

xml.etree.ElementTree.Element – An ElementTree object representation of the TRIPS EKB XML.

statements
list[indra.statements.Statement] – A list of INDRA Statements that were extracted from the EKB.

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

sentences
dict[str: str] – The list of all sentences in the EKB with their IDs

paragraphs
dict[str: str] – The list of all paragraphs in the EKB with their IDs

par_to_sec
dict[str: str] – A map from paragraph IDs to their associated section types

extracted_events
list[xml.etree.ElementTree.Element] – A list of Event elements that have been extracted as INDRA Statements.

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.

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*)
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.

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
list[str] – A list of EKB XML extractions corresponding to the input text list.

drum_system
subprocess.Popen – 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.

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.**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_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_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_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_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_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

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

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*)

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

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*, *num_documents=None*)

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**

str – The csxml file, containing Medscan XML, to process

indra.sources.medscan.api.**num_documents**

int – The number of documents to process, or None to process all of the documents within the csxml file.

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*)

ch_end
Alias for field number 5

ch_start
Alias for field number 4

name
Alias for field number 0

properties
Alias for field number 3

type
Alias for field number 2

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
list<str> – A list of extracted INDRA statements

sentence_statements
list<str> – 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.

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

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

last_site_info_in_sentence
SiteInfo – 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

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_from_file_handle(*f*, *num_documents*)

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

- **f** (*file object*) – A filehandle to a source of MedScan csxml data
- **num_documents** (*int*) – The number of documents to process, or None to process all documents in the input stream

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*)

name

Alias for field number 1

type

Alias for field number 0

urn

Alias for field number 2

class indra.sources.medscan.processor.**MedscanRelation**(*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.

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

sec
str – The section of the document the relation occurs in

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

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

subj
str – The entity ID of the subject

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

obj
str – The entity ID of the object

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

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.**is_statement_in_list**(*statement*, *statement_list*)

Return True of given statement is equivalent to on in a list

Determines whether the statement is equivalent to any statement in the given list of statements, with equivalency determined by Statement's equals method.

Parameters

- **statement** (indra.statements.Statement) – The statement to compare with
- **statement_list** (*list[indra.statements.Statement]*) – The statement list whose entries we compare with statement

Returns **in_list** – True if statement is equivalent to any statements in the list

Return type bool

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.**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

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.**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

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

list[indra.statements.Statement] – A list of INDRA statements extracted from the provided text via TEES

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. For processing nxml files, install the nxml2txt utility (https://github.com/spyysalo/nxml2txt) and set the configuration variable NXML2TXT_PATH to its location. In addition, since the reader works with Python 2 only, make sure PYTHON2_PATH is set in your config file or environment and points to a Python 2 executable.

ISI API (indra.sources.isi.api)

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

indra.sources.isi.api.**process_nxml**(*nxml_filename*, *pmid=None*, *extra_annotations=None*, *cleanup=True*, *add_grounding=True*)
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.
- **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

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*)

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

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

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

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

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

indra.sources.isi.api.**process_text**(*text*, *pmid=None*, *cleanup=True*, *add_grounding=True*)

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)
- **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

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=True*)

Processes the output of the ISI reader.

reader_output
json – The output JSON of the ISI reader as a json object.

verbs
set[str] – A list of verbs that have appeared in the processed ISI output

pmid
str – The PMID to assign to the extracted Statements

extra_annotations
dict – Annotations to be included with each extracted Statement

statements
list[indra.statements.Statement] – Extracted statements

get_statements()
Process reader output to produce INDRA Statements.

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_b*, *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
list[indra.statements.Statement] – A list of INDRA statements converted from Geneways action mentions, populated by calling the constructor

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

4.2.2 General Purpose Reading Systems

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 set up reading with Eidos, the Eidos system and its dependencies need to be compiled and 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 operational to use Eidos reading in Python. For more details, see *Pyjnius* setup instructions in the documentation.

For eidos to provide grounding information to be included in INDRA Statements, the eidos configuration needs to be adjusted. First, download vectors.txt from https://drive.google.com/open?id=1tffQuLB5XtKcq9wlo0n-tsnPJYQF18oS and put it in a folder called src/main/resources/org/clulab/wm/eidos/w2v within your eidos folder. Next, set the property "useW2V" to true in src/main/resources/eidos.conf. Finally, rerun sbt assembly.

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**(*json_dict*)
Return an EidosProcessor by processing a Eidos JSON-LD dict.

Parameters **json_dict** (*dict*) – The JSON-LD dict to be processed.

Returns **ep** – A EidosProcessor containing the extracted INDRA Statements in its statements attribute.

Return type *EidosProcessor*

indra.sources.eidos.api.**process_json_file**(*file_name*)
Return an EidosProcessor by processing the given Eidos JSON-LD file.

This function is useful if the output from Eidos is saved as a file and needs to be processed.

Parameters **file_name** (*str*) – The name of the JSON-LD file to be processed.

Returns **ep** – A EidosProcessor containing the extracted INDRA Statements in its statements attribute.

Return type *EidosProcessor*

indra.sources.eidos.api.**process_json_ld**(*json_dict*)
DEPRECATED: see process_json

indra.sources.eidos.api.**process_json_ld_file**(*file_name*)
DEPRECATED: see process_json_file

indra.sources.eidos.api.**process_json_ld_str**(*json_str*)
DEPRECATED: see process_json_str

indra.sources.eidos.api.**process_json_str**(*json_str*)
Return an EidosProcessor by processing the Eidos JSON-LD string.

Parameters **json_str** (*str*) – The JSON-LD string to be processed.

Returns **ep** – A EidosProcessor containing the extracted INDRA Statements in its statements attribute.

Return type *EidosProcessor*

indra.sources.eidos.api.**process_text**(*text*, *out_format='json_ld'*, *save_json='eidos_output.json'*, *webservice=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.
- **out_format** (*Optional[str]*) – The type of Eidos output to read into and process. Currently only 'json-ld' is supported which is also the default value used.
- **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

Returns **ep** – An EidosProcessor containing the extracted INDRA Statements in its statements attribute.

Return type *EidosProcessor*

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.

tree
objectpath.Tree – The objectpath Tree object representing the extractions.

statements
list[indra.statements.Statement] – A list of INDRA Statements that were extracted by the processor.

static find_arg(*event*, *arg_type*)
Return ID of the first argument of a given type

static find_args(*event*, *arg_type*)
Return IDs of all arguments of a given type

geo_context_from_ref(*ref*)
Return a ref context object given a location reference entry.

get_causal_relations()
Extract causal relations as Statements.

static get_concept(*entity*)
Return Concept from an Eidos entity.

get_evidence(*event*)
Return the Evidence object for the INDRA Statment.

static 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"}]

static ref_context_from_geoloc(*geoloc*)
Return a RefContext object given a geoloc entry.

static time_context_from_dct(*dct*)
Return a time context object given a DCT entry.

time_context_from_ref(*timex*)
Return a time context object given a timex reference entry.

static time_context_from_timex(*timex*)
Return a TimeContext object given a timex entry.

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
org.clulab.wm.eidos.EidosSystem – A Scala object, an instance of the Eidos reading system. It is instantiated only when first processing text.

process_text(*text*, *format='json'*)
Return a mentions JSON object given text.

Parameters

- **text** (*str*) – Text to be processed.
- **format** (*str*) – The format of the output to produce, one of "json" or "json_ld". Default: "json"

Returns **json_dict** – A JSON object of mentions extracted from text.

Return type dict

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*)
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

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.ExtractFromDirectory' will run 'org.clulab.wm.eidos.apps.ExtractFromDirectory'
- ***args** – Any further arguments to be passed as inputs to the class being run.

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.

CWMS (indra.sources.cwms)

CWMS is a variant of the TRIPS system. It is a general purpose natural language understanding system with applications in world modeling. For more information, see: http://trips.ihmc.us/parser/cgi/cwmsreader

CWMS API (indra.sources.cwms.api)

indra.sources.cwms.api.**process_ekb**(*ekb_str*)
Processes an EKB string produced by CWMS.

Parameters **ekb_str** (*str*) – EKB string to process

Returns **cp** – A CWMSProcessor, which contains a list of INDRA statements in its statements attribute.

Return type indra.sources.cwms.CWMSProcessor

indra.sources.cwms.api.**process_ekb_file**(*fname*)
Processes an EKB file produced by CWMS.

Parameters **fname** (*str*) – Path to the EKB file to process.

Returns **cp** – A CWMSProcessor, which contains a list of INDRA statements in its statements attribute.

Return type indra.sources.cwms.CWMSProcessor

indra.sources.cwms.api.**process_rdf_file**(*text*, *rdf_filename*)
Process CWMS's RDF output for the given statement and returns a processor populated with INDRA statements.

Parameters

- **text** (*str*) – Sentence to process
- **rdf_filename** (*str*) – The RDF filename to process

Returns **cp** – A CWMSProcessor instance, which contains a list of INDRA Statements as its statements attribute.

Return type indra.sources.cwms.CWMSRDFProcessor

indra.sources.cwms.api.**process_text**(*text*, *save_xml='cwms_output.xml'*)
Processes text using the CWMS web service.

Parameters **text** (*str*) – Text to process

Returns **cp** – A CWMSProcessor, which contains a list of INDRA statements in its statements attribute.

Return type indra.sources.cwms.CWMSProcessor

CWMS EKB Processor (indra.sources.cwms.processor)

exception indra.sources.cwms.processor.**CWMSError**

class indra.sources.cwms.processor.**CWMSProcessor**(*xml_string*)
The CWMSProcessor currently extracts causal relationships between terms (nouns) in EKB. In the future, this processor can be extended to extract other types of relations, or to extract relations involving events.

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
xml.etree.ElementTree.Element – An ElementTree object representation of the TRIPS EKB XML.

doc_id
str – Document ID

statements
list[indra.statements.Statement] – A list of INDRA Statements that were extracted from the EKB.

sentences
dict[str: str] – The list of all sentences in the EKB with their IDs

paragraphs
dict[str: str] – The list of all paragraphs in the EKB with their IDs

par_to_sec
dict[str: str] – A map from paragraph IDs to their associated section types

extract_noun_relations(*key*)
Extract relationships where a term/noun affects another term/noun

CWMS RDF Processor (indra.sources.cwms.rdf_processor)

class indra.sources.cwms.rdf_processor.**CWMSRDFProcessor**(*text*, *rdf_filename*)
This processor extracts INDRA statements from CWMS RDF output.

Parameters

- **text** (*str*) – The source sentence as text
- **rdf_filename** (*str*) – A string containing the RDF output returned by CWMS for that sentence

statements
list[indra.statements.Statement] – A list of INDRA statements that were extracted by the processor.

extract_statement_from_query_result(*res*)
Adds a statement based on one element of a rdflib SPARQL query.

Parameters **res** (*rdflib.query.ResultRow*) – Element of rdflib SPARQL query result

extract_statements()
Extracts INDRA statements from the RDF graph via SPARQL queries.

Sofia (indra.sources.sofia)

Sofia is a general purpose natural language processing system developed at UPitt and CMU by N. Miskov et al.

Sofia API (indra.sources.sofia.api)

indra.sources.sofia.api.**process_table**(*fname*)
Return processor by processing a given sheet of a spreadsheet file.

Parameters **fname** (*str*) – The name of the Excel file (typically .xlsx extension) to process

Returns **sp** – A SofiaProcessor object which has a list of extracted INDRA Statements as its statements attribute

Return type indra.sources.sofia.processor.SofiaProcessor

Sofia Processor (indra.sources.sofia.processor)

Hume (indra.sources.hume)

Hume is a general purpose reading system developed by BBN.

Currently, INDRA can process JSON-LD files produced by Hume. When available, the API will be extended with access to the reader as a service.

Hume API (indra.sources.hume.api)

indra.sources.hume.api.**process_jsonld_file**(*fname*)
Process a JSON-LD file in the new format to extract Statements.

Parameters **fname** (*str*) – The path to the JSON-LD file to be processed.

Returns A HumeProcessor instance, which contains a list of INDRA Statements as its statements attribute.

Return type indra.sources.hume.HumeProcessor

indra.sources.hume.api.**process_jsonld**(*jsonld*)
Process a JSON-LD string in the new format to extract Statements.

Parameters **jsonld** (*dict*) – The JSON-LD object to be processed.

Returns A HumeProcessor instance, which contains a list of INDRA Statements as its statements attribute.

Return type indra.sources.hume.HumeProcessor

Hume Processor (indra.sources.hume.processor)

class indra.sources.hume.processor.**HumeJsonLdProcessor**(*json_dict*)
This processor extracts INDRA Statements from Hume JSON-LD output.

Parameters **json_dict** (*dict*) – A JSON dictionary containing the Hume extractions in JSON-LD format.

tree
objectpath.Tree – The objectpath Tree object representing the extractions.

statements
list[indra.statements.Statement] – A list of INDRA Statements that were extracted by the processor.

get_documents()
Populate the sentences attribute with a dict keyed by document id.

4.2.3 Standard Molecular Pathway Databases

BEL (indra.sources.bel)

BEL API (indra.sources.bel.api)

indra.sources.bel.api.**process_belrdf**(*rdf_str*, *print_output=True*)
Return a BelRdfProcessor for a BEL/RDF string.

Parameters **rdf_str** (*str*) – A BEL/RDF string to be processed. This will usually come from reading a .rdf file.

Returns **bp** – A BelRdfProcessor object which contains INDRA Statements in bp.statements.

Return type *BelRdfProcessor*

Notes

This function calls all the specific get_type_of_mechanism() functions of the newly constructed BelRdfProcessor to extract INDRA Statements.

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 *identifier_validation* to False.

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_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_ndex_neighborhood**(*gene_names*, *network_id=None*, *rdf_out='bel_output.rdf'*, *print_output=True*)

Return a BelRdfProcessor for an NDEx network neighborhood.

Parameters

- **gene_names** (*list*) – A list of HGNC gene symbols to search the neighborhood of. Example: ['BRAF', 'MAP2K1']
- **network_id** (*Optional[str]*) – The UUID of the network in NDEx. By default, the BEL Large Corpus network is used.
- **rdf_out** (*Optional[str]*) – Name of the output file to save the RDF returned by the web service. This is useful for debugging purposes or to repeat the same query on an offline RDF file later. Default: bel_output.rdf

Returns **bp** – A BelRdfProcessor object which contains INDRA Statements in bp.statements.

Return type *BelRdfProcessor*

Notes

This function calls process_belrdf to the returned RDF string from the webservice.

indra.sources.bel.api.**process_pybel_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**(*gene_names*, *network_file=None*, *network_type='belscript'*)
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

- **network_file** (*Optional[str]*) – Path to the network file to process. If not given, by default, the BEL Large Corpus is used.
- **network_type** (*Optional[str]*) – This function allows processing both BEL Script files and JSON files. This argument controls which type is assumed to be processed, and the value can be either 'belscript' or 'json'. Default: bel_script

Returns **bp** – A PybelProcessor object which contains INDRA Statements in bp.statements.

Return type *PybelProcessor*

BEL RDF Processor (indra.sources.bel.rdf_processor)

class indra.sources.bel.rdf_processor.**BelRdfProcessor**(*g*)
The BelRdfProcessor extracts INDRA Statements from a BEL RDF model.

Parameters **g** (*rdflib.Graph*) – An RDF graph object containing the BEL model.

g
rdflib.Graph – An RDF graph object containing the BEL model.

statements
list[indra.statements.Statement] – A list of extracted INDRA Statements representing direct mechanisms. This list should be used for assembly in INDRA.

indirect_stmts
list[indra.statements.Statement] – A list of extracted INDRA Statements representing indirect mechanisms. This list should be used for assembly or model checking in INDRA.

converted_direct_stmts
list[str] – A list of all direct BEL statements, as strings, that were converted into INDRA Statements.

converted_indirect_stmts
list[str] – A list of all indirect BEL statements, as strings, that were converted into INDRA Statements.

degenerate_stmts
list[str] – A list of degenerate BEL statements, as strings, in the BEL model.

all_direct_stmts
list[str] – A list of all BEL statements representing direct interactions, as strings, in the BEL model.

all_indirect_stmts
list[str] – A list of all BEL statements that represent indirect interactions, as strings, in the BEL model.

get_activating_mods()
Extract INDRA ActiveForm Statements with a single mod from BEL.

The SPARQL pattern used for extraction from BEL looks for a ModifiedProteinAbundance as subject and an Activiy of a ProteinAbundance as object.

Examples

proteinAbundance(HGNC:INSR,proteinModification(P,Y)) directlyIncreases kinaseActivity(proteinAbundance(HGNC:INSR))

get_activating_subs()
Extract INDRA ActiveForm Statements based on a mutation from BEL.

The SPARQL pattern used to extract ActiveForms due to mutations look for a ProteinAbundance as a subject which has a child encoding the amino acid substitution. The object of the statement is an ActivityType of the same ProteinAbundance, which is either increased or decreased.

Examples

proteinAbundance(HGNC:NRAS,substitution(Q,61,K)) directlyIncreases gtpBoundActivity(proteinAbundance(HGNC:NRAS))

proteinAbundance(HGNC:TP53,substitution(F,134,I)) directlyDecreases transcriptionalActivity(proteinAbundance(HGNC:TP53))

get_activation()
Extract INDRA Inhibition/Activation Statements from BEL.

The SPARQL query used to extract Activation Statements looks for patterns in which the subject is is an ActivityType (of a ProtainAbundance) or an Abundance (of a small molecule). The object has to be the ActivityType (typically of a ProteinAbundance) which is either increased or decreased.

Examples

abundance(CHEBI:gefitinib) directlyDecreases kinaseActivity(proteinAbundance(HGNC:EGFR))

kinaseActivity(proteinAbundance(HGNC:MAP3K5)) directlyIncreases kinaseActivity(proteinAbundance(HGNC:MAP2K7))

This pattern covers the extraction of Gap/Gef and GtpActivation Statements, which are recognized by the object activty or the subject activity, respectively, being *gtpbound*.

Examples

catalyticActivity(proteinAbundance(HGNC:RASA1)) directlyDecreases gtpBoundActivity(proteinAbundance(PFH:"RAS Family"))

catalyticActivity(proteinAbundance(HGNC:SOS1)) directlyIncreases gtpBoundActivity(proteinAbundance(HGNC:HRAS))

gtpBoundActivity(proteinAbundance(HGNC:HRAS)) directlyIncreases catalyticActivity(proteinAbundance(HGNC:TIAM1))

get_all_direct_statements()
Get all directlyIncreases/Decreases BEL statements.

This method stores the results of the query in self.all_direct_stmts as a list of strings. The SPARQL query used to find direct BEL statements searches for all statements whose predicate is either DirectyIncreases or DirectlyDecreases.

get_all_indirect_statements()
Get all indirect increases/decreases BEL statements.

This method stores the results of the query in self.all_indirect_stmts as a list of strings. The SPARQL query used to find indirect BEL statements searches for all statements whose predicate is either Increases or Decreases.

get_complexes()
Extract INDRA Complex Statements from BEL.

The SPARQL query used to extract Complexes looks for ComplexAbundance terms and their constituents. This pattern is distinct from other patterns in this processor in that it queries for terms, not full statements.

Examples

complexAbundance(proteinAbundance(HGNC:PPARG), proteinAbundance(HGNC:RXRA)) decreases biologicalProcess(MESHPP:"Insulin Resistance")

get_conversions()
Extract Conversion INDRA Statements from BEL.

The SPARQL query used to extract Conversions searches for a subject (controller) which is an AbundanceActivity which directlyIncreases a Reaction with a given list of Reactants and Products.

Examples

catalyticActivity(proteinAbundance(HGNC:HMOX1)) directlyIncreases reaction(reactants(abundance(CHEBI:heme)), products(abundance(SCHEM:Biliverdine), abundance(CHEBI:"carbon monoxide")))

get_degenerate_statements()
Get all degenerate BEL statements.

Stores the results of the query in self.degenerate_stmts.

get_modifications()
Extract INDRA Modification Statements from BEL.

Two SPARQL patterns are used for extracting Modifications from BEL:

- q_phospho1 assumes that the subject is an AbundanceActivity, which increases/decreases a ModifiedProteinAbundance.

  Examples:

  kinaseActivity(proteinAbundance(HGNC:IKBKE)) directlyIncreases proteinAbundance(HGNC:IRF3,proteinModification(P,S,385))

  phosphataseActivity(proteinAbundance(HGNC:DUSP4)) directlyDecreases proteinAbundance(HGNC:MAPK1,proteinModification(P,T,185))

- q_phospho2 assumes that the subject is a ProteinAbundance which increases/decreases a ModifiedProteinAbundance.

  Examples:

  proteinAbundance(HGNC:NGF) increases proteinAbundance(HGNC:NFKBIA,proteinModification(P,Y,42))

  proteinAbundance(HGNC:FGF1) decreases proteinAbundance(HGNC:RB1,proteinModification(P))

get_transcription()
Extract Increase/DecreaseAmount INDRA Statements from BEL.

Three distinct SPARQL patterns are used to extract amount regulations from BEL.

- q_tscript1 searches for a subject which is a Transcription ActivityType of a ProteinAbundance and an object which is an RNAAbundance that is either increased or decreased.

  Examples:

  transcriptionalActivity(proteinAbundance(HGNC:FOXP2)) directlyIncreases rnaAbundance(HGNC:SYK)

  transcriptionalActivity(proteinAbundance(HGNC:FOXP2)) directlyDecreases rnaAbundance(HGNC:CALCRL)

- q_tscript2 searches for a subject which is a ProteinAbundance and an object which is an RNAAbundance. Note that this pattern typically exists in an indirect form (i.e. increases/decreases).

  Example:

  proteinAbundance(HGNC:MTF1) directlyIncreases rnaAbundance(HGNC:LCN1)

- q_tscript3 searches for a subject which is a ModifiedProteinAbundance, with an object which is an RNAAbundance. In the BEL large corpus, this pattern is found for subjects which are protein families or mouse/rat proteins, and the predicate in an indirect increase.

  Example:

  proteinAbundance(PFR:"Akt Family",proteinModification(P)) increases rnaAbundance(RGD:Cald1)

print_statement_coverage()
Display how many of the direct statements have been converted.

Also prints how many are considered 'degenerate' and not converted.

print_statements()
Print all extracted INDRA Statements.

indra.sources.bel.rdf_processor.**namespace_from_uri**(*uri*)
Return the entity namespace from the URI. Examples: http://www.openbel.org/bel/p_HGNC_RAF1 -> HGNC http://www.openbel.org/bel/p_RGD_Raf1 -> RGD http://www.openbel.org/bel/p_PFH_MEK1/2_Family -> PFH

indra.sources.bel.rdf_processor.**term_from_uri**(*uri*)
Removes prepended URI information from terms.

PyBEL Processor (indra.sources.bel.processor)

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
list[indra.statements.Statement] – A list of extracted INDRA Statements representing BEL Statements.

Biopax (indra.sources.biopax)

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*)
Returns a BiopaxProcessor for a BioPAX OWL file.

Parameters **owl_filename** (*string*) – The name of the 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 **bp** – A BiopaxProcessor containing the obtained BioPAX model in bp.model.

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*)

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
org.biopax.paxtools.model.Model – A BioPAX model object (java object) which is queried using Paxtools to extract INDRA Statements

statements
list[indra.statements.Statement] – A list of INDRA Statements that were extracted from the model.

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.

get_activity_modification()
Extract INDRA ActiveForm statements from the BioPAX model.

This method extracts ActiveForm Statements that are due to protein modifications. This method reuses the structure of BioPAX Pattern's org.biopax.paxtools.pattern.PatternBox.constrolsStateChange pattern with additional constraints to specify the gain or loss of a modification occurring (phosphorylation, deubiquitination, etc.) and the gain or loss of activity due to the modification state change.

get_complexes()
Extract INDRA Complex Statements from the BioPAX model.

This method searches for org.biopax.paxtools.model.level3.Complex objects which represent molecular complexes. It doesn't reuse BioPAX Pattern's org.biopax.paxtools.pattern.PatternBox.inComplexWith query since that retrieves pairs of complex members rather than the full complex.

get_conversions()
Extract Conversion INDRA Statements from the BioPAX model.

This method uses a custom BioPAX Pattern (one that is not implemented PatternBox) to query for BiochemicalReactions whose left and right hand sides are collections of SmallMolecules. This pattern thereby extracts metabolic conversions as well as signaling processes via small molecules (e.g. lipid phosphorylation or cleavage).

get_gap()
Extract Gap INDRA Statements from the BioPAX model.

This method uses a custom BioPAX Pattern (one that is not implemented PatternBox) to query for controlled BiochemicalReactions in which the same protein is in complex with GTP on the left hand side and in complex with GDP on the right hand side. This implies that the controller is a GAP for the GDP/GTP-bound protein.

get_gef()
Extract Gef INDRA Statements from the BioPAX model.

This method uses a custom BioPAX Pattern (one that is not implemented PatternBox) to query for controlled BiochemicalReactions in which the same protein is in complex with GDP on the left hand side and in complex with GTP on the right hand side. This implies that the controller is a GEF for the GDP/GTP-bound protein.

get_modifications()
Extract INDRA Modification Statements from the BioPAX model.

To extract Modifications, this method reuses the structure of BioPAX Pattern's org.biopax.paxtools.pattern.PatternBox.constrolsStateChange pattern with additional constraints to specify the type of state change occurring (phosphorylation, deubiquitination, etc.).

get_regulate_activities()
Get Activation/Inhibition INDRA Statements from the BioPAX model.

This method extracts Activation/Inhibition Statements and reuses the structure of BioPAX Pattern's org.biopax.paxtools.pattern.PatternBox.constrolsStateChange pattern with additional constraints to specify the gain or loss of activity state but assuring that the activity change is not due to a modification state change (which are extracted by get_modifications and get_activity_modification).

get_regulate_amounts()
Extract INDRA RegulateAmount Statements from the BioPAX model.

This method extracts IncreaseAmount/DecreaseAmount Statements from the BioPAX model. It fully reuses BioPAX Pattern's org.biopax.paxtools.pattern.PatternBox.controlsExpressionWithTemplateReac pattern to find TemplateReactions which control the expression of a protein.

print_statements()
Print all INDRA Statements collected by the processors.

save_model(*file_name=None*)
Save the BioPAX model object in an OWL file.

Parameters **file_name** (*Optional[str]*) – The name of the OWL file to save the model in.

Pathway Commons Client (indra.sources.biopax.pathway_commons_client)

indra.sources.biopax.pathway_commons_client.**graph_query**(*kind*, *source*, *target=None*, *neighbor_limit=1*, *database_filter=None*)
Perform a graph query on PathwayCommons.

For more information on these queries, see http://www.pathwaycommons.org/pc2/#graph

Parameters

- **kind** (*str*) – The kind of graph query to perform. Currently 3 options are implemented, 'neighborhood', 'pathsbetween' and 'pathsfromto'.
- **source** (*list[str]*) – A list of gene names which are the source set for the graph query.
- **target** (*Optional[list[str]]*) – A list of gene names which are the target set for the graph query. Only needed for 'pathsfromto' queries.
- **neighbor_limit** (*Optional[int]*) – This limits the length of the longest path considered in the graph query. Default: 1

Returns **model** – A BioPAX model (java object).

Return type org.biopax.paxtools.model.Model

indra.sources.biopax.pathway_commons_client.**model_to_owl**(*model*, *fname*)
Save a BioPAX model object as an OWL file.

Parameters

- **model** (*org.biopax.paxtools.model.Model*) – A BioPAX model object (java object).
- **fname** (*str*) – The name of the OWL file to save the model in.

indra.sources.biopax.pathway_commons_client.**owl_str_to_model**(*owl_str*)
Return a BioPAX model object from an OWL string.

Parameters **owl_str** (*str*) – The model as an OWL string.

Returns **biopax_model** – A BioPAX model object (java object).

Return type org.biopax.paxtools.model.Model

indra.sources.biopax.pathway_commons_client.**owl_to_model**(*fname*)
Return a BioPAX model object from an OWL file.

Parameters **fname** (*str*) – The name of the OWL file containing the model.

Returns **biopax_model** – A BioPAX model object (java object).

Return type org.biopax.paxtools.model.Model

SIGNOR (indra.sources.signor)

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
list[indra.statements.Statements] – Extracted INDRA Complex statements.

physical_only
boolean – Indicates whether only physical interactions were included during statement processing.

4.2.4 Custom Knowledge Bases

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.

TAS API (indra.sources.tas.api)

indra.sources.tas.api.**process_csv**(*affinity_class_limit=2*)
Return a TasProcessor for the contents of the csv contained in data.

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

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*)
A processor for the Target Affinity Spectrum data table.

LINCS Drug Targets (indra.sources.lincs_drug)

This module provides and API and processor for the HMS LINCS small molecule target relationship database. This is a manually curated set of relationships with the "nominal" target of each drug determined by a human expert. Note that the determination of the "nominal" target is not always backed up by experimentally measured affinities. The underlying data is available here: http://lincs.hms.harvard.edu/db/datasets/20000/results

LINCS Drug Targets API (indra.sources.lincs_drug.api)

indra.sources.lincs_drug.api.**process_from_web**()
Return a processor for the LINCS drug target data.

Returns A LincsProcessor object which contains extracted INDRA Statements in its statements attribute.

Return type *LincsProcessor*

LINCS Drug Targets Processor (indra.sources.lincs_drug.processor)

class indra.sources.lincs_drug.processor.**LincsProcessor**(*lincs_data*)
Processor for the HMS LINCS drug target dataset.

Parameters **lincs_data** (*list[dict]*) – A list of dicts with keys set by the header of the csv, and values from the data in the csv.

statements
list[indra.statements.Statement] – A list of indra statements extracted from the CSV file.

NDEx CX API (indra.sources.ndex_cx.api)

indra.sources.ndex_cx.api.**process_cx**(*cx_json*, *require_grounding=True*)
Process a CX JSON object into Statements.

Parameters

- **cx_json** (*list*) – CX JSON object.
- **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*, *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.

statements
list – A list of extracted INDRA Statements. Not all edges in the network may be converted into Statements.

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 possibly 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, you may contact the developers to request a URL and API key.

Client API (`indra.sources.indra_db_rest.client_api`)

indra.sources.indra_db_rest.client_api.**get_statements**(*subject=None*, *object=None*, *agents=None*, *stmt_type=None*, *use_exact_type=False*, *persist=True*, *timeout=None*, *simple_response=True*, *ev_limit=10*, *best_first=True*, *tries=2*, *max_stmts=None*)

Get Statements from the INDRA DB web API matching given agents and type.

There are two types of response available. You can just get a list of INDRA Statements, or you can get an IndraRestResponse object, which allows 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.

*In the sense of having the most supporting evidence.

Parameters

- **subject/object** (*str*) – Optionally specify the subject and/or object of the statements in 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.

- **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. If simple_response is True, all statements available will be returned. Otherwise (if None), block indefinitely until all statements are retrieved. Default is None.
- **simple_response** (*bool*) – If True, a simple list of statements is returned (thus block should also be True). If block is False, only the original sample will be returned (as though persist was False), until the statements are done loading, in which case the rest should appear in the list. This behavior is not encouraged. Default is True (for the sake of backwards compatibility).
- **ev_limit** (*int or None*) – Limit the amount of evidence returned per Statement. Default is 10.
- **best_first** (*bool*) – If True, the preassembled statements will be sorted by the amount of evidence they have, and those with the most evidence will be prioritized. When using *max_stmts*, this means you will get the "best" statements. If False, statements will be queried in arbitrary order.
- **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 2.
- **max_stmts** (*int or None*) – Select the maximum number of statements to return. When set less than 1000 the effect is much the same as setting persist to false, and will guarantee a faster response. Default is None.

Returns **stmts** – A list of INDRA Statement instances. Note that if a supporting or supported Statement was not included in your query, it will simply be instantiated as an *Unresolved* statement, with *uuid* of the statement.

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

indra.sources.indra_db_rest.client_api.**get_statements_for_paper**(*ids*, *ev_limit=10*, *best_first=True*, *tries=2*, *max_stmts=None*)

Get the set of raw Statements extracted from a paper given by the id.

Parameters

- **ids** (*list[(<id type>, <id value>)]*) – A list of tuples with ids and their type. 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.
- **ev_limit** (*int or None*) – Limit the amount of evidence returned per Statement. Default is 10.
- **best_first** (*bool*) – If True, the preassembled statements will be sorted by the amount of evidence they have, and those with the most evidence will be prioritized. When using *max_stmts*, this means you will get the "best" statements. If False, statements will be queried in arbitrary order.

- **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 2.
- **max_stmts** (*int or None*) – Select a maximum number of statements to be returned. Default is None.

Returns **stmts** – A list of INDRA Statement instances.

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

indra.sources.indra_db_rest.client_api.**get_statements_by_hash**(*hash_list*, *ev_limit=100*, *best_first=True*, *tries=2*)

Get fully formed statements from a list of hashes.

Parameters

- **hash_list** (*list[int or str]*) – A list of statement hashes.
- **ev_limit** (*int or None*) – Limit the amount of evidence returned per Statement. Default is 100.
- **best_first** (*bool*) – If True, the preassembled statements will be sorted by the amount of evidence they have, and those with the most evidence will be prioritized. When using *max_stmts*, this means you will get the "best" statements. If False, statements will be queried in arbitrary order.
- **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 2.

indra.sources.indra_db_rest.client_api.**submit_curation**(*hash_val*, *tag*, *curator*, *text=None*, *source='indra_rest_client'*, *ev_hash=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** (*str*) – The name or identifier for 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*.
- **is_test** (*bool*) – Used in testing. If True, no curation will actually be added to the database.

exception indra.sources.indra_db_rest.client_api.**IndraDBRestAPIError**(*resp*)

exception indra.sources.indra_db_rest.client_api.**IndraDBRestClientError**

exception indra.sources.indra_db_rest.client_api.**IndraDBRestResponseError**

4.3 Database clients (indra.databases)

4.3.1 HGNC client (indra.hgnc_client)

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_hgnc_entry**
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_entrez**(*entrez_id*)
Return the HGNC ID corresponding to the given Entrez ID.

Parameters **entrez_id** (*str*) – The EntrezC ID to be converted, a number passed as a strig.

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

4.3.2 Uniprot client (indra.databases.uniprot_client)

indra.databases.uniprot_client.**get_family_members**(*family_name*, *human_only=True*)
Return the HGNC gene symbols which are the members of a given family.

Parameters

- **family_name** (*str*) – Family name to be queried.
- **human_only** (*bool*) – If True, only human proteins in the family will be returned. Default: True

Returns **gene_names** – The HGNC gene symbols corresponding to the given family.

Return type list

indra.databases.uniprot_client.**get_function**(*protein_id*)
Return the function description of a given protein.

Parameters **protein_id** (*str*) – The UniProt ID of the protein.

Returns The function description of the protein.

Return type str

indra.databases.uniprot_client.**get_gene_name**(*protein_id*, *web_fallback=True*)
Return the gene name for the given UniProt ID.

This is an alternative to get_hgnc_name and is useful when HGNC name is not availabe (for instance, when the organism is not homo sapiens).

Parameters

- **protein_id** (*str*) – UniProt ID to be mapped.
- **web_fallback** (*Optional[bool]*) – If True and the offline lookup fails, the UniProt web service is used to do the query.

Returns **gene_name** – The gene name corresponding to the given Uniprot ID.

Return type str

indra.databases.uniprot_client.**get_gene_synonyms**(*protein_id*)

Return a list of synonyms for the gene corresponding to a protein.

Note that synonyms here also include the official gene name as returned by get_gene_name.

Parameters **protein_id** (*str*) – The UniProt ID of the protein to query

Returns **synonyms** – The list of synonyms of the gene corresponding to the protein

Return type list[str]

indra.databases.uniprot_client.**get_id_from_mgi**(*mgi_id*)

Return the UniProt ID given the MGI ID of a mouse protein.

Parameters **mgi_id** (*str*) – The MGI ID of the mouse protein.

Returns **up_id** – The UniProt ID of the mouse protein.

Return type str

indra.databases.uniprot_client.**get_id_from_mnemonic**(*uniprot_mnemonic*)

Return the UniProt ID for the given UniProt mnemonic.

Parameters **uniprot_mnemonic** (*str*) – UniProt mnemonic to be mapped.

Returns **uniprot_id** – The UniProt ID corresponding to the given Uniprot mnemonic.

Return type str

indra.databases.uniprot_client.**get_id_from_rgd**(*rgd_id*)

Return the UniProt ID given the RGD ID of a rat protein.

Parameters **rgd_id** (*str*) – The RGD ID of the rat protein.

Returns **up_id** – The UniProt ID of the rat protein.

Return type str

indra.databases.uniprot_client.**get_length**(*protein_id*)

Return the length (number of amino acids) of a protein.

Parameters **protein_id** (*str*) – UniProt ID of a protein.

Returns **length** – The length of the protein in amino acids.

Return type int

indra.databases.uniprot_client.**get_mgi_id**(*protein_id*)

Return the MGI ID given the protein id of a mouse protein.

Parameters **protein_id** (*str*) – UniProt ID of the mouse protein

Returns **mgi_id** – MGI ID of the mouse protein

Return type str

indra.databases.uniprot_client.**get_mnemonic**(*protein_id*, *web_fallback=False*)
Return the UniProt mnemonic for the given UniProt ID.

Parameters

- **protein_id** (*str*) – UniProt ID to be mapped.
- **web_fallback** (*Optional[bool]*) – If True and the offline lookup fails, the UniProt web service is used to do the query.

Returns **mnemonic** – The UniProt mnemonic corresponding to the given Uniprot ID.

Return type str

indra.databases.uniprot_client.**get_mouse_id**(*human_protein_id*)
Return the mouse UniProt ID given a human UniProt ID.

Parameters **human_protein_id** (*str*) – The UniProt ID of a human protein.

Returns **mouse_protein_id** – The UniProt ID of a mouse protein orthologous to the given human protein

Return type str

indra.databases.uniprot_client.**get_primary_id**(*protein_id*)
Return a primary entry corresponding to the UniProt ID.

Parameters **protein_id** (*str*) – The UniProt ID to map to primary.

Returns **primary_id** – If the given ID is primary, it is returned as is. Othwewise the primary IDs are looked up. If there are multiple primary IDs then the first human one is returned. If there are no human primary IDs then the first primary found is returned.

Return type str

indra.databases.uniprot_client.**get_protein_synonyms**(*protein_id*)
Return a list of synonyms for a protein.

Note that this function returns protein synonyms as provided by UniProt. The get_gene_synonym returns synonyms given for the gene corresponding to the protein, and get_synonyms returns both.

Parameters **protein_id** (*str*) – The UniProt ID of the protein to query

Returns **synonyms** – The list of synonyms of the protein

Return type list[str]

indra.databases.uniprot_client.**get_rat_id**(*human_protein_id*)
Return the rat UniProt ID given a human UniProt ID.

Parameters **human_protein_id** (*str*) – The UniProt ID of a human protein.

Returns **rat_protein_id** – The UniProt ID of a rat protein orthologous to the given human protein

Return type str

indra.databases.uniprot_client.**get_rgd_id**(*protein_id*)
Return the RGD ID given the protein id of a rat protein.

Parameters **protein_id** (*str*) – UniProt ID of the rat protein

Returns **rgd_id** – RGD ID of the rat protein

Return type str

indra.databases.uniprot_client.**get_synonyms**(*protein_id*)
Return synonyms for a protein and its associated gene.

Parameters **protein_id** (*str*) – The UniProt ID of the protein to query

Returns **synonyms** – The list of synonyms of the protein and its associated gene.

Return type list[str]

indra.databases.uniprot_client.**is_human**(*protein_id*)
Return True if the given protein id corresponds to a human protein.

Parameters **protein_id** (*str*) – UniProt ID of the protein

Returns

Return type True if the protein_id corresponds to a human protein, otherwise False.

indra.databases.uniprot_client.**is_mouse**(*protein_id*)
Return True if the given protein id corresponds to a mouse protein.

Parameters **protein_id** (*str*) – UniProt ID of the protein

Returns

Return type True if the protein_id corresponds to a mouse protein, otherwise False.

indra.databases.uniprot_client.**is_rat**(*protein_id*)
Return True if the given protein id corresponds to a rat protein.

Parameters **protein_id** (*str*) – UniProt ID of the protein

Returns

Return type True if the protein_id corresponds to a rat protein, otherwise False.

indra.databases.uniprot_client.**is_secondary**(*protein_id*)
Return True if the UniProt ID corresponds to a secondary accession.

Parameters **protein_id** (*str*) – The UniProt ID to check.

Returns

Return type True if it is a secondary accessing entry, False otherwise.

indra.databases.uniprot_client.**query_protein**
Return the UniProt entry as an RDF graph for the given UniProt ID.

Parameters **protein_id** (*str*) – UniProt ID to be queried.

Returns **g** – The RDF graph corresponding to the UniProt entry.

Return type rdflib.Graph

indra.databases.uniprot_client.**verify_location**(*protein_id*, *residue*, *location*)
Return True if the residue is at the given location in the UP sequence.

Parameters

- **protein_id** (*str*) – UniProt ID of the protein whose sequence is used as reference.
- **residue** (*str*) – A single character amino acid symbol (Y, S, T, V, etc.)
- **location** (*str*) – The location on the protein sequence (starting at 1) at which the residue should be checked against the reference sequence.

Returns

- *True if the given residue is at the given position in the sequence*
- *corresponding to the given UniProt ID, otherwise False.*

indra.databases.uniprot_client.**verify_modification**(*protein_id*, *residue*, *location=None*)

Return True if the residue at the given location has a known modifiation.

Parameters

- **protein_id** (*str*) – UniProt ID of the protein whose sequence is used as reference.
- **residue** (*str*) – A single character amino acid symbol (Y, S, T, V, etc.)
- **location** (*Optional[str]*) – The location on the protein sequence (starting at 1) at which the modification is checked.

Returns

- *True if the given residue is reported to be modified at the given position*
- *in the sequence corresponding to the given UniProt ID, otherwise False.*
- *If location is not given, we only check if there is any residue of the*
- *given type that is modified.*

4.3.3 ChEBI client (indra.databases.chebi_client)

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.
- **chebi_id** (*str*) – The ChEBI ID corresponding to the given CAS ID. If the lookup fails, None is returned.

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_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_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

4.3.4 BioGRID client (indra.databases.biogrid_client)

indra.databases.biogrid_client.**get_publications**(*gene_names*, *save_json_name=None*)
Return evidence publications for interaction between the given genes.

Parameters

- **gene_names** (*list[str]*) – A list of gene names (HGNC symbols) to query interactions between. Currently supports exactly two genes only.
- **save_json_name** (*Optional[str]*) – A file name to save the raw BioGRID web service output in. By default, the raw output is not saved.

Returns **publications** – A list of Publication objects that provide evidence for interactions between the given list of genes.

Return type list[Publication]

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 Network relevance client (indra.databases.relevance_client)

indra.databases.relevance_client.**get_heat_kernel**(*network_id*)
Return the identifier of a heat kernel calculated for a given network.

Parameters **network_id** (*str*) – The UUID of the network in NDEx.

Returns **kernel_id** – The identifier of the heat kernel calculated for the given network.

Return type str

indra.databases.relevance_client.**get_relevant_nodes**(*network_id*, *query_nodes*)
Return a set of network nodes relevant to a given query set.

A heat diffusion algorithm is used on a pre-computed heat kernel for the given network which starts from the given query nodes. The nodes in the network are ranked according to heat score which is a measure of relevance with respect to the query nodes.

Parameters

- **network_id** (*str*) – The UUID of the network in NDEx.
- **query_nodes** (*list[str]*) – A list of node names with respect to which relevance is queried.

Returns **ranked_entities** – A list containing pairs of node names and their relevance scores.

Return type list[(str, float)]

4.3.7 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

4.3.8 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**

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

4.3.9 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_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

4.3.10 LINCS client (indra.databases.lincs_client)

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]

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.**load_lincs_csv**(*url*)
Helper function to turn csv rows into dicts.

4.3.11 MeSH client (indra.databases.mesh_client)

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**
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

4.3.12 GO client (indra.databases.go_client)

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.**load_go_graph**(*go_fname*)
Load the GO data from an OWL file and parse into an RDF graph.

Parameters **go_fname** (*str*) – Path to the GO OWL file. Can be downloaded from http://geneontology.org/ontology/go.owl.

Returns RDF graph containing GO data.

Return type rdflib.Graph

indra.databases.go_client.**update_id_mappings**(*g*)
Compile all ID->label mappings and save to a TSV file.

Parameters **g** (*rdflib.Graph*) – RDF graph containing GO data.

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]*) – 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**

Get the XML metadata for a single article from the Pubmed database.

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 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**

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** (*string*) – 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_issns_for_journal**

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_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 PMIDs as strings*) – Can contain 1-200 PMIDs.
- **get_issns_from_nlm** (*boolean*) – 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** (*boolean*) – Indicates whether to include the Pubmed abstract in the results.
- **prepend_title** (*boolean*) – 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*)

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** (*boolean*) – 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** (*boolean*) – Indicates whether to include the Pubmed abstract in the results.
- **prepend_title** (*boolean*) – 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_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.**filter_pmids**(*pmid_list*, *source_type*)
Filter a list of PMIDs for ones with full text from PMC.

Parameters

- **pmid_list** (*list*) – List of PMIDs to filter.
- **source_type** (*string*) – One of 'fulltext', 'oa_xml', 'oa_txt', or 'auth_xml'.

Returns

Return type list of PMIDs available in the specified source/format type.

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 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**
Returns the metadata of an article given its DOI from CrossRef as a JSON dict

4.4.4 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_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**
Search ScienceDirect through the API for articles.

See http://api.elsevier.com/content/search/fields/scidir for constructing a query string to pass here. Example: 'abstract(BRAF) AND all("colorectal cancer")'

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**
Search ScienceDirect with a query string constrained to a given year.

Parameters

- **query_str** (*str*) – The query string to search with
- **date** (*str*) – The year to constrain the search to

Returns **piis** – The list of PIIs identifying the papers returned by the search

Return type list[str]

4.4.5 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

4.5 Preassembly (indra.preassembler)

4.5.1 Preassembler (indra.preassembler)

class indra.preassembler.**Preassembler**(*hierarchies*, *stmts=None*)
De-duplicates statements and arranges them in a specificity hierarchy.

Parameters

- **hierarchies** (dict[*indra.preassembler.hierarchy_manager*]) – A dictionary of hierarchies with keys such as 'entity' (hierarchy of entities, primarily specifying relationships between genes and their families) and 'modification' pointing to HierarchyManagers
- **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.

stmts
list of *indra.statements.Statement* – Starting set of statements for preassembly.

unique_stmts
list of *indra.statements.Statement* – Statements resulting from combining duplicates.

related_stmts
list of *indra.statements.Statement* – Top-level statements after building the refinement hierarchy.

hierarchies
dict[*indra.preassembler.hierarchy_manager*] – A dictionary of hierarchies with keys such as 'entity' and 'modification' pointing to HierarchyManagers

add_statements(*stmts*)
Add to the current list of statements.

Parameters **stmts** (list of *indra.statements.Statement*) – Statements to add to the current list.

static 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:

```
>>> 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')])
>>> uniq_stmts = Preassembler.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 static method *combine_duplicate_stmts()*.

combine_related(*return_toplevel=True*, *poolsize=None*, *size_cutoff=100*)
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 grouped by type (e.g., Phosphorylation) and each type is iterated over independently.
2. Statements of the same type are then grouped according to their Agents' entity hierarchy component identifiers. For instance, ERK, MAPK1 and MAPK3 are all in the same connected component in the entity hierarchy and therefore all Statements of the same type referencing these entities will be grouped. This grouping assures that relations are only possible within Statement groups and not among groups. For two Statements to be in the same group at this step, the Statements must be the same type and the Agents at each position in the Agent lists must either be in the same hierarchy

component, or if they are not in the hierarchy, must have identical entity_matches_keys. Statements with None in one of the Agent list positions are collected separately at this stage.

3. Statements with None at either the first or second position are iterated over. For a statement with a None as the first Agent, the second Agent is examined; then the Statement with None is added to all Statement groups with a corresponding component or entity_matches_key in the second position. The same procedure is performed for Statements with None at the second Agent position.
4. The statements within each group are then compared; if one 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.
5. 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.

On multi-core machines, the algorithm can be parallelized by setting the poolsize argument to the desired number of worker processes. This feature is only available in Python > 3.4.

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
- **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.

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.preassembler.hierarchy_manager import hierarchies
>>> braf = Agent('BRAF')
>>> map2k1 = Agent('MAP2K1')
```

(continues on next page)

(continued from previous page)

```
>>> st1 = Phosphorylation(braf, map2k1)
>>> st2 = Phosphorylation(braf, map2k1, residue='S')
>>> pa = Preassembler(hierarchies, [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*))

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.preassembler.hierarchy_manager import hierarchies
>>> 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(hierarchies, [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.preassembler.hierarchy_manager import hierarchies
>>> braf = Agent('BRAF')
>>> map2k1 = Agent('MAP2K1')
>>> st1 = Phosphorylation(braf, map2k1)
>>> st2 = Phosphorylation(braf, map2k1, residue='S')
>>> pa = Preassembler(hierarchies, [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

- **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().
- **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.
- **rank_dir** (*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.

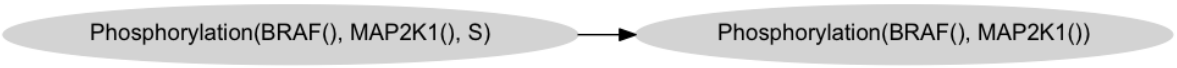
Return type pygraphviz.AGraph

Examples

Pattern for getting statements and rendering as a Graphviz graph:

```
>>> from indra.preassembler.hierarchy_manager import hierarchies
>>> braf = Agent('BRAF')
>>> map2k1 = Agent('MAP2K1')
>>> st1 = Phosphorylation(braf, map2k1)
>>> st2 = Phosphorylation(braf, map2k1, residue='S')
>>> pa = Preassembler(hierarchies, [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:



4.5.2 Entity grounding curation and mapping (indra.preassembler.grounding_mapper)

class indra.preassembler.grounding_mapper.**GroundingMapper**(*gm*, *agent_map=None*)

Maps grounding of INDRA Agents based on a given grounding map.

gm

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.

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** (*indra.statements.Agent*) – The grounded Agent.
- **maps_to_none** (*bool*) – True if the Agent is in the grounding map and maps to None.

map_agents(*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*

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*

rename_agents(*stmts*)

Return a list of mapped statements with updated agent names.

Creates a new list of statements without modifying the original list.

The agents in a statement should be renamed if the grounding map has updated their db_refs. 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. If neither a FamPlex ID or HGNC name can be found, falls back to the original name.

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*

update_agent_db_refs(*agent*, *agent_text*, *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
- **agent_text** (*str*) – The agent_text to find a grounding for in the grounding map dictionary. Typically this will be agent.db_refs['TEXT'] but there may be situations where a different value should be used.
- **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

Raises

- `ValueError` – If the the grounding map contains and HGNC symbol for agent_text but no HGNC ID can be found for it.
- `ValueError` – If the grounding map contains both an HGNC symbol and a Uniprot ID, but the HGNC symbol and the gene name associated with the gene in Uniprot do not match or if there is no associated gene name in Uniprot.

indra.preassembler.grounding_mapper.**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.**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.**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.**get_agents_with_name**(*name*, *stmts*)

Return all agents within a list of statements with a particular name.

indra.preassembler.grounding_mapper.**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.**load_grounding_map**(*grounding_map_path*, *ignore_path=None*, *lineterminator='\r\n'*)

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.

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>
- **ignore_path** (*Optional[str]*) – Path to csv file containing terms that should be filtered out during the grounding mapping process. The file Should be of the form <agent_text>,,..., where the number of commas that appear is the same as in the csv file at grounding_map_path. Default: None
- **lineterminator** (*Optional[str]*) – Line terminator used in input csv file. Default:

Returns **g_map** – The grounding map constructed from the given files.

Return type dict

indra.preassembler.grounding_mapper.**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.**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.**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.**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

4.5.3 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 tuples*) – A list of invalid sites, where each entry in the list has two elements: ((gene_name, residue, position), mapped_site). If the invalid position was not found in the site map, mapped_site is None; otherwise it is a tuple consisting of (residue, position, comment). Note that some entries in the site map are curated *errors*, that is, sites that are known to be frequent misattributions to certain proteins. Such sites are mapped to tuples (None, None, comment).
- **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*, *use_cache=False*)
Use curated site information to standardize modification sites in stmts.

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

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: (('MAP2K1', 'S', '217'), ('S', '218', 'off by one'))
                 (('MAP2K1', 'S', '221'), ('S', '222', 'off by one'))
                 (('MAPK1', 'T', '183'), ('T', '185', 'off by two; mouse sequence
↪'))
    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
[(('MAP2K1', 'S', '217'), ('S', '218', 'off by one')), (('MAP2K1', 'S', '221'), (
↪'S', '222', 'off by one')), (('MAPK1', 'T', '183'), ('T', '185', 'off by two;␣
↪mouse sequence'))]
>>> ms.mapped_stmt
Phosphorylation(MAP2K1(mods: (phosphorylation, S, 218), (phosphorylation, S,␣
↪222)), MAPK1(), T, 185)
```

map_sites(*stmts*, *do_methionine_offset=True*, *do_orthology_mapping=True*, *do_isoform_mapping=True*)

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.
- **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.

Returns 2-tuple containing (valid_statements, mapped_statements). The first element of the tuple is a list 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

indra.preassembler.sitemapper.**default_mapper = <indra.preassembler.sitemapper.SiteMapper ob**
A default instance of *SiteMapper* that contains the site information found in resources/curated_site_map.csv'.

indra.preassembler.sitemapper.**load_site_map**(*path*)
Load the modification site map from a file.

The site map file should be a comma-separated file with six columns:

```
Gene: HGNC gene name
OrigRes: Original (incorrect) residue
OrigPos: Original (incorrect) residue position
CorrectRes: The correct residue for the modification
CorrectPos: The correct residue position
Comment: Description of the reason for the error.
```

Parameters **path** (*string*) – Path to the tab-separated site map file.

Returns 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.).

Return type dict

4.5.4 Hierarchy manager (indra.preassembler.hierarchy_manager)

class indra.preassembler.hierarchy_manager.**HierarchyManager**(*rdf_file*, *build_closure=True*, *uri_as_name=True*)
Store hierarchical relationships between different types of entities.

Used to store, e.g., entity hierarchies (proteins and protein families) and modification hierarchies (serine phosphorylation vs. phosphorylation).

Parameters

- **rdf_file** (*string*) – Path to the RDF file containing the hierarchy.
- **build_closure** (*Optional[bool]*) – If True, the transitive closure of the hierarchy is generated up from to speed up processing. Default: True

- **uri_as_name** (*Optional[bool]*) – If True, entries are accessed directly by their URIs. If False entries are accessed by finding their name through the hasName relationship. Default: True

graph
instance of *rdflib.Graph* – The RDF graph containing the hierarchy.

build_transitive_closure(*rel*, *tc_dict*)
Build a transitive closure for a given relation in a given dict.

build_transitive_closures()
Build the transitive closures of the hierarchy.

This method constructs dictionaries which contain terms in the hierarchy as keys and either all the "isa+" or "partof+" related terms as values.

directly_or_indirectly_related(*ns1*, *id1*, *ns2*, *id2*, *closure_dict*, *relation_func*)
Return True if two entities have the speicified relationship.

This relation is constructed possibly through multiple links connecting the two entities directly or indirectly.

Parameters

- **ns1** (*str*) – Namespace code for an entity.
- **id1** (*str*) – URI for an entity.
- **ns2** (*str*) – Namespace code for an entity.
- **id2** (*str*) – URI for an entity.
- **closure_dict** (*dict*) – A dictionary mapping node names to nodes that have the specified relationship, directly or indirectly. Empty if this has not been precomputed.
- **relation_func** (*function*) – Function with arguments (node, graph) that generates objects with some relationship with node on the given graph.

Returns True if t1 has the specified relationship with t2, either directly or through a series of intermediates; False otherwise.

Return type bool

extend_with(*rdf_file*)
Extend the RDF graph of this HierarchyManager with another RDF file.

Parameters **rdf_file** (*str*) – An RDF file which is parsed such that the current graph and the graph described by the file are merged.

find_entity
Get the entity that has the specified name (or synonym).

Parameters **x** (*string*) – Name or synonym for the target entity.

get_children(*uri*)
Return all (not just immediate) children of a given entry.

Parameters **uri** (*str*) – The URI of the entry whose children are to be returned. See the get_uri method to construct this URI from a name space and id.

get_parents(*uri*, *type='all'*)
Return parents of a given entry.

Parameters

- **uri** (*str*) – The URI of the entry whose parents are to be returned. See the get_uri method to construct this URI from a name space and id.
- **type** (*str*) – 'all': return all parents irrespective of level; 'immediate': return only the immediate parents; 'top': return only the highest level parents

is_opposite(*ns1*, *id1*, *ns2*, *id2*)
Return True if two entities are in an "is_opposite" relationship

Parameters

- **ns1** (*str*) – Namespace code for an entity.
- **id1** (*str*) – URI for an entity.
- **ns2** (*str*) – Namespace code for an entity.
- **id2** (*str*) – URI for an entity.

Returns True if t1 has an "is_opposite" relationship with t2.

Return type bool

isa(*ns1*, *id1*, *ns2*, *id2*)
Return True if one entity has an "isa" relationship to another.

Parameters

- **ns1** (*str*) – Namespace code for an entity.
- **id1** (*string*) – URI for an entity.
- **ns2** (*str*) – Namespace code for an entity.
- **id2** (*str*) – URI for an entity.

Returns True if t1 has an "isa" relationship with t2, either directly or through a series of intermediates; False otherwise.

Return type bool

isa_or_partof(*ns1*, *id1*, *ns2*, *id2*)
Return True if two entities are in an "isa" or "partof" relationship

Parameters

- **ns1** (*str*) – Namespace code for an entity.
- **id1** (*str*) – URI for an entity.
- **ns2** (*str*) – Namespace code for an entity.
- **id2** (*str*) – URI for an entity.

Returns True if t1 has a "isa" or "partof" relationship with t2, either directly or through a series of intermediates; False otherwise.

Return type bool

partof(*ns1*, *id1*, *ns2*, *id2*)
Return True if one entity is "partof" another.

Parameters

- **ns1** (*str*) – Namespace code for an entity.
- **id1** (*str*) – URI for an entity.
- **ns2** (*str*) – Namespace code for an entity.

- **id2** (*str*) – URI for an entity.

Returns True if t1 has a "partof" relationship with t2, either directly or through a series of intermediates; False otherwise.

Return type bool

exception indra.preassembler.hierarchy_manager.**UnknownNamespaceException**

4.6 Belief Engine (indra.belief)

class indra.belief.**BeliefEngine**(*scorer=None*)

Assigns beliefs to INDRA Statements based on supporting evidence.

scorer

BeliefScorer – A BeliefScorer object that computes the prior probability of a statement given its its statment type and evidence. Must implement the *score_statement* 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.

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 on its prior probability and the probabilities propagated from Statements supporting it in the hierarchy graph.

Parameters **statements** (*list[indra.statements.Statement]*) – A list of INDRA Statements whose belief scores are to be calculated. Each Statement object's belief attribute is updated by this function.

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[indra.mechlinker.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.

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** (*list[indra.statements.Statement]*) – A list of INDRA Statements whose belief scores are to be calculated. Each Statement object's belief attribute is updated by this function.

class indra.belief.**BeliefPackage**(*statement_key*, *evidences*)

evidences

Alias for field number 1

statement_key
Alias for field number 0

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** (*list<indra.statements.Statement>*) – List of statements to check

score_statement(*st*, *extra_evidence=None*)
Computes the prior belief probability for an INDRA Statement.

The Statement is assumed to be de-duplicated. In other words, the Statement is assumed to have a list of Evidence objects that supports it. The prior probability of the Statement is calculated based on the number of Evidences it has and their sources.

Parameters

- **st** (indra.statements.Statement) – An INDRA Statements whose belief scores are to be calculated.
- **extra_evidence** (*list[indra.statements.Evidence]*) – A list of Evidences that are supporting the Statement (that aren't already included in the Statement's own evidence list.

Returns **belief_score** – The computed prior probability for the statement

Return type float

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** (*dict[dict]*) – 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** (*dict[dict]*) – 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** (*list[indra.statements.Statement]*) – List of statements to check

score_statement (*st*, *extra_evidence=None*)
Computes the prior belief probability for an INDRA Statement.

The Statement is assumed to be de-duplicated. In other words, the Statement is assumed to have a list of Evidence objects that supports it. The prior probability of the Statement is calculated based on the number of Evidences it has and their sources.

Parameters

- **st** (indra.statements.Statement) – An INDRA Statements whose belief scores are to be calculated.
- **extra_evidence** (*list[indra.statements.Evidence]*) – A list of Evidences that are supporting the Statement (that aren't already included in the Statement's own evidence list.

Returns **belief_score** – The computed prior probability for the statement

Return type float

indra.belief.**evidence_random_noise_prior** (*evidence*, *type_probs*, *subtype_probs*)
Determines 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.

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 Staement with a belief score of 0.7 will end up in the returned Statement list with probability 0.7.

Parameters

- **stmts** (*list[indra.statements.Statement]*) – A list of INDRA Statements to sample.
- **seed** (*Optional[int]*) – A seed for the random number generator used for sampling.

Returns **new_stmts** – A list of INDRA Statements that were chosen by random sampling according to their respective belief scores.

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

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** (indra.statements.Evidence) – The statement which we wish to subtype

Returns **types** – A tuple with (type, subtype), both strings Returns (type, None) if the type of statement is not yet handled in this function.

Return type tuple

4.7 Mechanism Linker (indra.mechlinker)

class indra.mechlinker.**AgentState** (*agent*, *evidence=None*)
A class representing Agent state without identifying a specific Agent.

bound_conditions
list[indra.statements.BoundCondition]

mods
list[indra.statements.ModCondition]

mutations
list[indra.statements.Mutation]

location
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*]

4.8 Assemblers of model output (`indra.assemblers`)

4.8.1 Executable PySB models (`indra.assemblers.pysb.assembler`)

class indra.assemblers.pysb.assembler.**PysbAssembler**(*policies=None*)
Assembler creating a PySB model from a set of INDRA Statements.

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'}

policies
dict – A dictionary of policies that defines assembly policies for Statement types. It is assigned in the constructor.

statements
list – A list of INDRA statements to be assembled.

model
pysb.Model – A PySB model object that is assembled by this class.

agent_set
_BaseAgentSet – A set of BaseAgents used during the assembly process.

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

Return type str

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 of policies, as defined in `indra.assemblers.PysbAssembler`. 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 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.**export_sbgn**(*model*)
Return an SBGN model string corresponding to the PySB model.

This function first calls generate_equations on the PySB model to obtain a reaction network (i.e. individual species, reactions). It then iterates over each reaction and and instantiates its reactants, products, and the process itself as SBGN glyphs and arcs.

Parameters **model** (*pysb.core.Model*) – A PySB model to be exported into SBGN

Returns **sbgn_str** – An SBGN model as string

Return type str

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 http://identifiers.org/.

indra.assemblers.pysb.assembler.**get_binding_site_name**(*agent*)
Return a binding site name from a given agent.

indra.assemblers.pysb.assembler.**get_create_parameter**(*model*, *name*, *value*, *unique=True*)
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.

indra.assemblers.pysb.assembler.**get_mod_site_name**(*mod_type*, *residue*, *position*)
Return site names for a modification.

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*)
Get monomer patterns for the agent accounting for grounding information.

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.

4.8.2 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
list[indra.statements.Statement] – A list of INDRA Statements to be assembled.

network_name
str – The name of the network to be assembled.

cx
dict – The structure of the CX network that is assembled.

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

4.8.3 Natural language (indra.assemblers.english.assembler)

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
list[indra.statements.Statement] – A list of INDRA Statements to assemble.

model
str – The assembled sentences as a single string.

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

4.8.4 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
list[indra.statements.Statement] – A list of INDRA Statements to be assembled.

graph
pygraphviz.AGraph – A pygraphviz graph that is assembled by this assembler.

existing_nodes
list[tuple] – The list of nodes (identified by node key tuples) that are already in the graph.

existing_edges
list[tuple] – The list of edges (identified by edge key tuples) that are already in the graph.

graph_properties
dict[str: str] – A dictionary of graphviz graph properties used for assembly.

node_properties
dict[str: str] – A dictionary of graphviz node properties used for assembly.

edge_properties
dict[str: str] – 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.

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

4.8.5 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 ineraction 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
networkx.DiGraph – A networkx graph that is assembled by this assembler.

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.

4.8.6 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
list – A list of INDRA statements to be assembled.

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

4.8.7 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
list[indra.statements.Statement] – A list of INDRA Statements to be assembled.

sbgn
lxml.etree.ElementTree – The structure of the SBGN model that is assembled, represented as an XML 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

4.8.8 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
list[indra.statements.Statement] – A list of INDRA Statements to be assembled.

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.

4.8.9 Causal analysis graphs (indra.assemblers.cag.assembler)

class indra.assemblers.cag.assembler.**CAGAssembler**(*stmts=None*)
Assembles a causal analysis graph from INDRA Statements.

Parameters **stmts** (*Optional[list[indra.statement.Statements]]*) – A list of INDRA Statements to be assembled. Currently supports Influence Statements.

statements
list[indra.statements.Statement] – A list of INDRA Statements to be assembled.

CAG
nx.MultiDiGraph – A networkx MultiDiGraph object representing the causal analysis graph.

add_statements(*stmts*)
Add a list of Statements to the assembler.

export_to_cytoscapejs()
Return CAG in format readable by CytoscapeJS.

Returns A JSON-like dict representing the graph for use with CytoscapeJS.

Return type dict

generate_jupyter_js(*cyjs_style=None*, *cyjs_layout=None*)
Generate Javascript from a template to run in Jupyter notebooks.

Parameters

- **cyjs_style** (*Optional[dict]*) – A dict that sets CytoscapeJS style as specified in https://github.com/cytoscape/cytoscape.js/blob/master/documentation/md/style.md.
- **cyjs_layout** (*Optional[dict]*) – A dict that sets CytoscapeJS layout parameters.

Returns A Javascript string to be rendered in a Jupyter notebook cell.

Return type str

make_model(*grounding_ontology='UN'*, *grounding_threshold=None*)
Return a networkx MultiDiGraph representing a causal analysis graph.

Parameters

- **grounding_ontology** (*Optional[str]*) – The ontology from which the grounding should be taken (e.g. UN, FAO)
- **grounding_threshold** (*Optional[float]*) – Minimum threshold score for Eidos grounding.

Returns The assembled CAG.

Return type nx.MultiDiGraph

4.8.10 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
list[indra.statements.Statement] – A list of INDRA Statements to be assembled.

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.

4.8.11 HTML browsing and curation (indra.assemblers.html.assembler)

Format a set of INDRA Statements into an HTML-formatted report which also supports curation.

class indra.assemblers.html.assembler.**HtmlAssembler**(*statements=None*, *summary_metadata=None*, *ev_totals=None*, *title='INDRA Results'*, *db_rest_url=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.
- **ev_totals** (*Optional[dict]*) – A dictionary of the total evidence available for each statement indexed by hash. Default: None
- **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

statements
list[indra.statements.Statement] – A list of INDRA Statements to assemble.

model
str – The HTML report formatted as a single string.

metadata
dict – Dictionary of statement list metadata such as that provided by the INDRA REST API.

ev_totals
dict – A dictionary of the total evidence available for each statement indexed by hash.

db_rest_url
str – The URL to a DB REST API.

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.

append_warning(*msg*)
Append a warning message to the model to expose issues.

make_model()
Return the assembled HTML content as a string.

Returns The assembled HTML as a string.

Return type str

save_model(*fname*)
Save the assembled HTML into a file.

Parameters **fname** (*str*) – The path to the file to save the HTML into.

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

4.8.12 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 Explanation (indra.explanation)

4.9.1 Check whether a rule-based model satisfies a property (indra.explanation.model_checker)

class indra.explanation.model_checker.**ModelChecker**(*model*, *statements=None*, *agent_obs=None*, *do_sampling=False*, *seed=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).

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*)
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

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*)
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

Returns True if the model satisfies the Statement.

Return type boolean

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_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.

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_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

class indra.explanation.model_checker.**PathMetric**(*source_node*, *target_node*, *polarity*, *length*)
Describes results of simple path search (path existence).

source_node
str – The source node of the path

target_node
str – The target node of the path

polarity
int – The polarity of the path between source and target

length
int – The length of the path

class indra.explanation.model_checker.**PathResult**(*path_found*, *result_code*, *max_paths*, *max_path_length*)
Describes results of running the ModelChecker on a single Statement.

path_found
bool – True if a path was found, False otherwise.

result_code
string –

- *STATEMENT_TYPE_NOT_HANDLED* - The provided statement type is not handled
- *SUBJECT_MONOMERS_NOT_FOUND* - Statement subject not found in model
- *OBSERVABLES_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

max_paths
int – The maximum number of specific paths to return for each Statement to be explained.

max_path_length
int – The maximum length of specific paths to return.

path_metrics
list[*indra.explanation.model_checker.PathMetric*] – A list of PathMetric objects, each describing the results of a simple path search (path existence).

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.

indra.explanation.model_checker.**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

indra.explanation.model_checker.**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*

4.10 Tools (indra.tools)

4.10.1 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*, *fname*)
Dump a list of statements into a pickle file.

Parameters **fname** (*str*) – The name of the pickle file to dump statements into.

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_db_refs**(*stmts_in*, *namespace*, *values*, *policy*, ***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*, ***kwargs*)

Filter to a given statement type.

Parameters

- **stmts_in** (*list[indra.statements.Statement]*) – A list of statements to filter.
- **stmt_type** (*indra.statements.Statement*) – The class of the statement type to filter for. Example: indra.statements.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_concept_names**(*stmts_in*, *name_list*, *policy*, ***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*, ***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*, ***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*, ***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*, ***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_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*, ***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.**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_grounding**(*stmts_in*, ***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).
- **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.**map_sequence**(*stmts_in*, ***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.**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_preassembly**(*stmts_in*, ***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** (*instance of indra.belief.BeliefScorer*) – Instance of BeliefScorer class to use in calculating Statement probabilities. If None is provided (default), then the default scorer is used.
- **hierarchies** (*dict*) – Dict of hierarchy managers to use for preassembly
- **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
- **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.
- **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.**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*]

4.10.2 Build a network from a gene list (indra.tools.gene_network)

class indra.tools.gene_network.**GeneNetwork**(*gene_list*, *basename=None*, *bel_corpus=None*)

Build a set of INDRA statements for a given gene list from databases.

Parameters

- **gene_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.
- **bel_corpus** (*str*) – Path to a BEL corpus to use. The default uses the BEL Large Corpus.

gene_list
str – List of gene names

basename
str or None – Filename prefix for cached intermediates, or None if no cached used.

results
dict – Dict containing results of preassembly (see return type for *run_preassembly()*.

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

4.10.3 Build an executable model from a fragment of a large network (indra.tools.executable_subnetwork)

indra.tools.executable_subnetwork.**get_subnetwork**(*statements*, *nodes*, *relevance_network=None*, *relevance_node_lim=10*)
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.
- **relevance_network** (*Optional[str]*) – The UUID of the NDEx network in which nodes relevant to the given nodes are found.
- **relevance_node_lim** (*Optional[int]*) – The maximal number of additional nodes to add to the subnetwork based on relevance.

Returns **model** – A PySB model object assembled using INDRA's PySB Assembler from the INDRA Statements corresponding to the subnetwork.

Return type pysb.Model

4.10.4 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

dict[str, list[indra.statements.Statement]] – A dictionary of INDRA Statements keyed by PMIDs that stores the current state of the IncrementalModel.

assembled_stmts

list[indra.statements.Statement] – A list of INDRA Statements after assembly.

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

4.10.5 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.

4.11 Util (indra.util)

4.11.1 Utilities for using AWS (indra.util.aws)

4.11.2 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.

4.11.3 A simple utility to get graphs from kappa (indra.util.kappa_util)

indra.util.kappa_util.**cm_json_to_graph**(*im_json*)
Return pygraphviz Agraph from Kappy's contact map JSON.

Parameters **im_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.util.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

4.11.4 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.

4.11.5 Some 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 5

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 0x7f62a1db8b38>
```

At this point *pa.model* contains a PySB model object with 3 monomers,

```
In [9]: for monomer in pa.model.monomers:
   ...:     print(monomer)
   ...:
Monomer('MAPK1', ['phospho', 'map2k', 'dusp'], {'phospho': ['u', 'p']})
Monomer('MAP2K1', ['mapk'])
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_
↪dephosphorylation_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)
```

(continues on next page)

(continued from previous page)

```
Parameter('kr_dm_bind_1', 0.1)
Parameter('kc_dm_dephosphorylation_1', 100.0)
Parameter('MAPK1_0', 10000.0)
Parameter('MAP2K1_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(MAPK1, 'http://identifiers.org/uniprot/P28482', 'is')
Annotation(MAPK1, 'http://identifiers.org/ncit/C21227', 'is')
Annotation(MAPK1, 'http://identifiers.org/hgnc/HGNC:6871', 'is')
Annotation(MAP2K1, 'http://identifiers.org/uniprot/Q02750', 'is')
Annotation(MAP2K1, 'http://identifiers.org/ncit/C21222', 'is')
Annotation(MAP2K1, 'http://identifiers.org/hgnc/HGNC:6840', 'is')
Annotation(DUSP6, 'http://identifiers.org/uniprot/Q16828', 'is')
Annotation(DUSP6, 'http://identifiers.org/ncit/C106026', 'is')
Annotation(DUSP6, 'http://identifiers.org/hgnc/HGNC:3072', 'is')
Annotation(MAP2K1_phosphorylation_bind_MAPK1_phospho, 'a8f29128-90e4-4379-8dfe-
↪3b89feb8469f', '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, 'a8f29128-90e4-4379-8dfe-3b89feb8469f
↪', 'from_indra_statement')
Annotation(MAP2K1_dissoc_MAPK1, 'a8f29128-90e4-4379-8dfe-3b89feb8469f', 'from_indra_
↪statement')
Annotation(DUSP6_dephosphorylation_bind_MAPK1_phospho, '002e2ced-a780-4c73-8997-
↪7305073ac992', '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, '002e2ced-a780-4c73-8997-
↪7305073ac992', 'from_indra_statement')
Annotation(DUSP6_dissoc_MAPK1, '002e2ced-a780-4c73-8997-7305073ac992', '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.

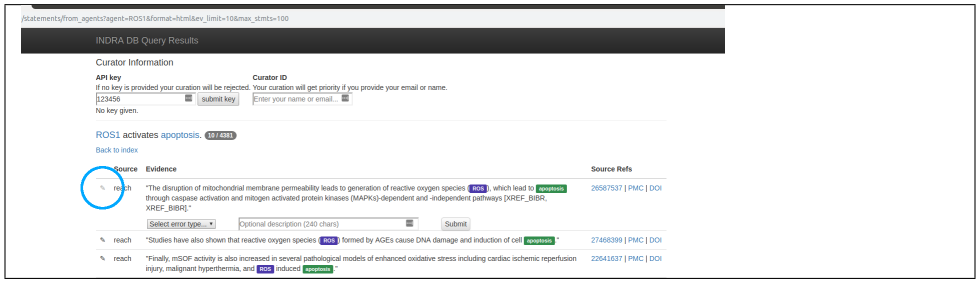
For more details about what options are available when doing curation, please refer to the curation section of the documentation.

5.2.1 Curating a Statement

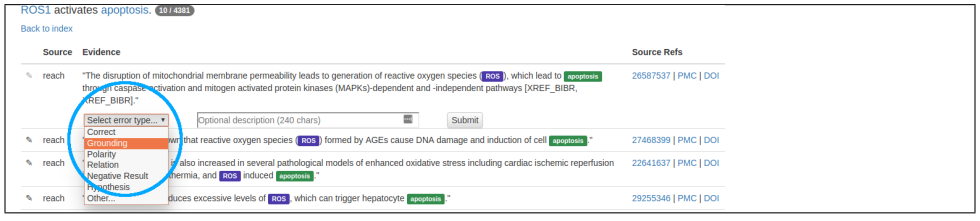
Let's assume you want to check any statements were ROS1 is an agent for errors. Let's also limit the number of statements to 100 and the number of evidences per statements to 10. This will speed up the query and page loading. The appropriate address to enter in your browser would then be:

```
http://api.host/statements/from_agents?agent=ROS1&format=html&ev_limit=10&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**, there are three minimum requirements:

1. A valid **API key** (at the top of the page, see image below)
2. A **curator ID**, such as name or email (at the top of the page, see image below)
3. A **selection in the dropdown menu** (by the curated statement)

If you selected *other...* in the dropdown menu, you must *also* describe the error in the textbox.

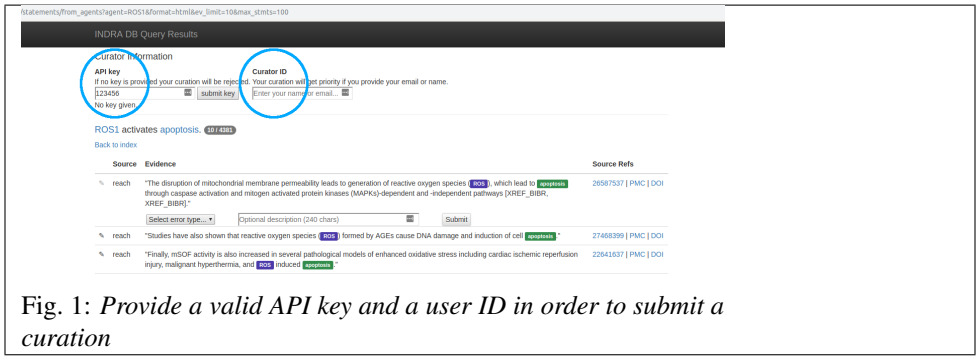
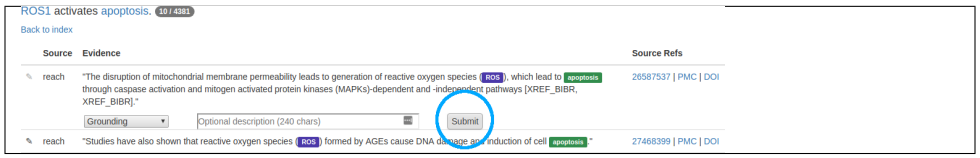


Fig. 1: *Provide a valid API key and a user ID in order to submit a curation*

When you have entered the necessary information, click the **Submit button** by the statement that you curated:



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. 2: *A green icon indicates a successfully submitted curation*

while a **red** indicates something went wrong with the submission:

Fig. 3: *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 Large-Scale Machine Reading with Starcluster

The following doc describes the steps involved in reading a large numbers of papers in parallel on Amazon EC2 using REACH, caching the JSON output on Amazon S3, then processing the REACH output into INDRA Statements. Prerequisites for doing the following are:

- A cluster of Amazon EC2 nodes configured using Starcluster, with INDRA installed and in the PYTHONPATH
- An Amazon S3 bucket containing full text contents for papers, keyed by Pubmed ID (creation of this S3 repository will be described in another tutorial).

This tutorial goes through the individual steps involved before describing how all of them can be run through the use of a single submission script, submit_reading_pipeline.py.

Note also that the prerequisite installation steps can be streamlined by putting them in a setup script that can be re-run upon instantiating a new Amazon cluster or by using them to configure a custom Amazon EC2 AMI.

5.3.1 Install REACH

Install SBT. On an EC2 Linux machine, run the following lines (drawn from http://www.scala-sbt.org/0.13/docs/Installing-sbt-on-Linux.html):

```
echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/
↪sbt.list
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 642AC823
sudo apt-get update
sudo apt-get install sbt
```

Clone REACH from https://github.com/clulab/reach.

Add the following line to reach/build.sbt:

```
mainClass in assembly := Some("org.clulab.reach.ReachCLI")
```

This assigns ReachCLI as the main class.

Compile and assemble REACH. Note that the path to the .ivy2 directory must be given. Use the assembly task to assemble a fat JAR containing all of the dependencies with the correct main class. Run the following from the directory containing the REACH build.sbt file (e.g., /pmc/reach).:

```
sbt -Dsbt.ivy.home=/pmc/reach/.ivy2 compile
sbt -Dsbt.ivy.home=/pmc/reach/.ivy2 assembly
```

5.3.2 Install Amazon S3 support

Install boto3:

```
pip install boto3
```

Note: If using EC2, make sure to install boto3, jsonpickle, and Amazon credentials on all nodes, not just the master node.

Add Amazon credentials to access the S3 bucket. First create the .aws directory on the EC2 instance:

```
mkdir /home/sgeadmin/.aws
```

Then set up Amazon credentials, for example by copying from your local machine using StarCluster:

```
starcluster put mycluster ~/.aws/credentials /home/sgeadmin/.aws
```

5.3.3 Install other dependencies

```
pip install jsonpickle # Necessary to process JSON from S3
pip install --upgrade pyjnius # Necessary for REACH
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-amd64
```

5.3.4 Assemble a Corpus of PMIDs

The first step in large-scale reading is to put together a file containing relevant Pubmed IDs. The simplest way to do this is to use the Pubmed search API to find papers associated with particular gene names, biological processes, or other search terms.

For example, to assemble a list of papers for SOS2 curated in Entrez Gene that are available in the Pubmed Central Open Access subset:

```
In [1]: from indra.literature import *

# Pick an example gene
In [2]: gene = 'SOS2'

# Get a list of PMIDs for the gene
In [3]: pmids = pubmed_client.get_ids_for_gene(gene)

# Get the PMIDs that have XML in PMC
In [4]: pmids_oa_xml = pmc_client.filter_pmids(pmids, 'oa_xml')

# Write the results to a file
In [5]: with open('%s_pmids.txt' % gene, 'w') as f:
   ...:     for pmid in pmids_oa_xml:
   ...:         f.write('%s\n' % pmid)
   ...:
```

This creates a file, SOS2_pmids.txt, containing the PMIDs that we will read with REACH.

5.3.5 Process the papers with REACH

The next step is to read the content of the papers with REACH in a parallelizable, high-throughput way. To do this, run the script indra/tools/reading/run_reach_on_pmids.py. If necessary update the lines at the top of the script with the REACH settings, e.g.:

```
cleanup = False
verbose = True
path_to_reach = '/pmc/reach/target/scala-2.11/reach-assembly-1.3.2-SNAPSHOT.jar'
reach_version = '1.3.2'
source_text = 'pmc_oa_xml'
force_read = False
```

The reach_version is important because it is used to determine whether the paper has already been read with this version of REACH (in which case it will be skipped), or if the REACH output needs to be updated. Alternatively, if you want to read all the papers regardless of whether they've been read before with the given version of REACH, set the force_read variable to True.

Next, create a top-level temporary directory to use during reading. This will be used to store the input files and the JSON output:

```
mkdir my_temp_dir
```

Run run_reach_on_pmids.py, passing arguments for the PMID list file, the temp directory, the number of cores to use on the machine, the PMID start index (in the PMID list file) and the end index. The start and end indices are used to subdivide the job into parallelizable chunks. If the end index is greater than the total number of PMIDs, it will process up to the last one in the list. For example:

```
python run_reach_on_pmids.py SOS2_pmids.txt my_temp_dir 8 0 10
```

This uses 8 cores to process the first ten papers listed in the file SOS2_pmids.txt. REACH will run, output the JSON files in the temporary directory, e.g. in my_temp_dir/read_0_to_10_MSP6YI/output, assemble the JSON files together, and upload the results to S3. If you attempt to process the files again with the same version of REACH, the script will detect that the JSON output from that version is already on S3 and skip those papers.

This can be submitted to run offline using the job scheduler on EC2 with, e.g.:

```
qsub -b y -cwd -V -pe orte 8 python run_reach_on_pmids.py SOS2_pmids.txt my_temp_dir␣
↪8 0 10
```

Note: The number of cores requested in the qsub call ('-pe orte 8') should match the number of cores passed to the run_reach_on_pmids.py script, which determines the number of threads that REACH will attempt to use (the third-to-last argument above). This should also match the total number of nodes on the Amazon EC2 node (e.g., 8 cores for c3.2xlarge). This way the job scheduler will schedule the job to run on all the cores of a single EC2 node, and REACH will use them all.

5.3.6 Extract INDRA Statements from the REACH output on S3

The script indra/tools/reading/process_reach_from_s3.py is used to extract INDRA Statements from the REACH output uploaded to S3 in the previous step. This process can also be parallelized by submitting chunks of papers to be processed by different cores. The INDRA statements for each chunk of papers are pickled and can be assembled into a single pickle file in a subsequent step.

Following the example above, run the following to process the REACH output for the SOS2 papers into INDRA statements. We'll do this in two chunks to show how the process can be parallelized and the statements assembled from multiple files:

```
python process_reach_from_s3.py SOS2_pmids.txt 0 5
python process_reach_from_s3.py SOS2_pmids.txt 5 10
```

The two runs create two different files for the results from the seven papers, reach_stmts_0_5.pkl (with statements from the first five papers) and reach_stmts_5_7.pkl (with statements from the last two). Note that the results are pickled as a dict (rather than a list), with PMIDs as keys and lists of Statements as values.

Of course, what we really want is a single file containing all of the statements for the entire corpus. To get this, run:

```
python assemble_reach_stmts.py reach_stmts_*.pkl
```

The results will be stored in reach_stmts.pkl.

5.3.7 Running the whole pipeline with one script

If you want to run the whole pipeline in one go, you can run the script submit_reading_pipeline.py (in indra/tools/reading) on a cluster of Amazon EC2 nodes. The script divides up the jobs evenly among the nodes and cores. Usage:

```
python submit_reading_pipeline.py pmid_list tmp_dir num_nodes num_cores_per_node
```

For example if you have a cluster with 8 c3.8xlarge nodes with 32 VCPUs each, you would call it with:

```
python submit_reading_pipeline.py SOS2_pmids.txt my_tmp_dir 8 32
```

The script submits the jobs to the scheduler with appropriate dependencies such that the REACH reading step completes first, then the INDRA processing step, and then the final assembly into a single pickle file.

5.4 Large-Scale Machine Reading with Amazon Batch

The following doc describes the steps involved in reading a large numbers of papers in parallel on Amazon EC2 using REACH, caching the JSON output on Amazon S3, processing the REACH output into INDRA Statements, and then caching the statements also on S3. Prerequisites for doing the following are:

- An Amazon S3 bucket containing full text contents for papers, keyed by Pubmed ID (creation of this S3 repository will be described in another tutorial).
- Amazon AWS credentials for using AWS Batch.
- A corpus of PMIDs (see *Large-Scale Machine Reading with Starcluster* for information on how to assemble this)
- Optional: Elsevier text and data mining API key and institution key for subscriber access to Elsevier full text content.

5.4.1 How it Works

- The reading pipeline makes use of a Docker image that contains INDRA and all necessary dependencies, including REACH, Kappa, PySB, etc. The Docker file for this image is available at: https://github.com/johnbachman/indra_docker.

- The INDRA Docker image is built by AWS Codebuild and pushed to Amazon's EC2 Container Service (ECS), where it is available via the Repository URI:

```
292075781285.dkr.ecr.us-east-1.amazonaws.com/indra
```

- An AWS Batch *Compute Environment* named "run_reach" is configured to use this Docker image for handling AWS jobs. This compute environment is configured to use only Spot instances with a maximum spot price of 40% of the on-demand price, and 16 vCPUs.
- An AWS *Job Queue*, "run_reach_queue", is configured to use instances of the "run_reach" Compute Environment.
- An AWS *Job Definition*, "run_reach_jobdef", is configured to run in the "run_reach_queue", and to use 16 vCPUs and 30GiB of RAM.
- Reading jobs are submitted by running the script:

```
python -m indra.tools.reading.submit_reading_pipeline_aws read [args]
```

  which, given a list of PMIDs:

  - Copies the PMID list to the key `reading_results/[job_name]/pmids` on Amazon S3
  - Breaks the list up into chunks (e.g., of 3000 PMIDs) and submits an AWS Batch job for each (using the "run_reach_jobdef" definition as a template).

- The ECS instance created by the AWS Batch job runs the script `indra.tools.reading.run_reach_on_pmids_aws`, which:
  - Checks for cached content on Amazon S3
  - If the PMID has not been read by the current version of REACH, checks for content
  - If the content is not available, downloads the content using the INDRA literature client, and caches on S3
  - The content to be read is downloaded to the `/tmp` directory of the instance
  - REACH is run using the command-line interface (RunReachCLI), and configured to read the papers in the `/tmp` directory using all of the vCPUs on the instance
  - When done, the result REACH JSON in the output folder is uploaded to S3
  - The JSON for both the previously and newly read papers is processed in parallel to INDRA Statements
  - The resulting subset of statements for the given range of papers is cached on S3 at `reading_results/[job_name]/stmts/[start_ix]_[end_ix].pkl`. This set of statements takes the form of a pickled (protocol 3) Python dict with PMIDs as keys and lists of INDRA Statements as values.
  - In addition, information about the sources of content available for each PMID is cached for each PMID subset at `reading_results/[job_name]/content_types/[start_ix]_[end_ix].pkl`.

- When the reading jobs for each of the subsets of PMIDs have been completed and cached on S3, the final combined set of statements (and combined information on content sources) can be assembled using:

```
python -m indra.tools.reading.submit_reading_pipeline_aws combine [job_name]
```

  - This script submits an AWS batch job for a machine with 1 vCPU but a large amount of memory (60GiB)
  - The job runs the script `indra.tools.reading.assemble_reach_stmts_aws`, which unpickles the results from all of the PMID subsets, combines them, and stores them on S3
  - The resulting files are obtainable from S3 at `reading_results/[job_name]/stmts.pkl` and `reading_results/[job_name]/content_types.pkl`.

- To run the entire pipeline, where the assembly of the combined set of statements is automatically performed after the reading step is completed, run:

```
python -m indra.tools.reading.submit_reading_pipeline_aws full [args]
```

5.5 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 example, assume that the gene of interest is TMEM173.

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!

5.5.1 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(['TMEM173'])
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.

5.5.2 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('TMEM173')
```

The variable *pmid* now contains a list of PMIDs associated with the gene.

5.5.3 Get the full text or abstract corresponding to the publications

Next we use INDRA's literature client to fetch the full text (if available) or the abstract corresponding to the PMIDs we have just collected.

```
from indra import literature

paper_contents = {}
for pmid in pmids:
```

(continues on next page)

(continued from previous page)

```
    content, content_type = literature.get_full_text(pmid, 'pmid')
    paper_contents[pmid] = (content, content_type)
```

We now have a dictionary called *paper_contents* which stores the content and the content type of each PMID we looked up.

5.5.4 Read the content of the publications

We next run the REACH reading system on the publications. Depending on the content type, different calls need to be made via INDRA's REACH API.

```
from indra import literature
from indra.sources import reach

read_offline = True

literature_stmts = []
for pmid, (content, content_type) in paper_contents.items():
    rp = None
    print('Reading %s' % pmid)
    if content_type == 'abstract':
        rp = reach.process_text(content, citation=pmid, offline=read_offline)
    elif content_type == 'pmc_oa_xml':
        rp = reach.process_nxml_str(content, offline=read_offline)
    elif content_type == 'elsevier_xml':
        txt = literature.elsevier_client.extract_text(content)
        if txt:
            rp = reach.process_text(txt, citation=pmid, offline=read_offline)
    if rp is not None:
        literature_stmts += rp.statements
```

The list *literature_stmts* now contains the results of all the statements that were read.

5.5.5 Combine all statements and run pre-assembly

```
from indra.tools import assemble_corpus

stmts = biopax_stmts + bel_stmts + literature_stmts

stmts = assemble_corpus.map_grounding(stmts)
stmts = assemble_corpus.map_sequence(stmts)
stmts = assemble_corpus.run_preassembly(stmts)
```

At this point *stmts* contains a list of Statements collected with grounding, sequences having been mapped, duplicates combined and less specific variants of statements hidden. It is possible to run other filters on the results such as to keep only human genes, remove Statements with ungrounded genes, or to keep only certain types of interactions.

5.5.6 Assemble the statements into a network model

```
from indra.assemblers.cx import CxAssembler
from indra.databases import ndex_client
```

(continues on next page)

(continued from previous page)

```
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)
```

CHAPTER 6

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 meant to be used locally (on a single machine or local network) and is currently not offered as a public web service by the creators of INDRA.

6.1 Installation

The REST service requires the *bottle* package to be installed in addition to all the other requirements of INDRA.

6.2 Launching the REST service

The REST service can be launched by running *api.py* in the *rest_api* folder within *indra*.

6.3 Documentation

The specific end-points and input/output parameters offered by the REST API are documented in *rest_api/docs/index.html*, which is accessible locally within the *indra* folder.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index

C

D

E

F

G

H

I

L

M

Q

R

S

T

U

V

W