

---

# **INDRA Documentation**

*Release 1.4.2*

**B. M. Gyori, J. A. Bachman**

**Sep 12, 2017**



<b>1</b>	<b>License and funding</b>	<b>3</b>
1.1	Installation	3
1.1.1	Installing Python	3
1.1.2	Installing INDRA	3
1.1.3	INDRA dependencies	4
1.2	Getting started with INDRA	6
1.2.1	Importing INDRA and its modules	6
1.2.2	Basic usage examples	6
1.3	INDRA modules reference	8
1.3.1	INDRA Statements ( <code>indra.statements</code> )	8
1.3.2	Processors for model input ( <code>indra.sources</code> )	20
1.3.3	Database clients ( <code>indra.databases</code> )	34
1.3.4	Literature clients ( <code>indra.literature</code> )	40
1.3.5	Preassembly ( <code>indra.preassembler</code> )	43
1.3.6	Belief Engine ( <code>indra.belief</code> )	52
1.3.7	Mechanism Linker ( <code>indra.mechlinker</code> )	53
1.3.8	Assemblers of model output ( <code>indra.assemblers</code> )	56
1.3.9	Explanation ( <code>indra.explanation</code> )	58
1.3.10	Tools ( <code>indra.tools</code> )	59
1.4	Tutorials	69
1.4.1	Using natural language to build models	69
1.4.2	Large-Scale Machine Reading with Starcluster	73
1.4.3	Large-Scale Machine Reading with Amazon Batch	77
1.4.4	Assembling everything known about a particular gene	78
1.5	REST API	80
1.5.1	Installation	80
1.5.2	Launching the REST service	80
1.5.3	Documentation	81
<b>2</b>	<b>Indices and tables</b>	<b>83</b>
	<b>Python Module Index</b>	<b>85</b>



INDRA (the Integrated Network and Dynamical Reasoning Assembler) assembles information about biochemical mechanisms into a common format that can be used to build several different kinds of explanatory models. 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 mechanistic *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.



INDRA is made available under the 2-clause [BSD license](#). Users are asked to acknowledge DARPA grant W911NF-14-1-0397, “Programmatic modelling for reasoning across complex mechanisms,” Peter Sorger and Dexter Pratt PIs.

Contents:

## Installation

### 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).

### Installing INDRA

#### 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.

### Cloning the source code from Github

You may want to simply clone the source code without installing INDRA as a system-wide package. In addition to cloning from Github, you need to run two git commands to update submodules in the INDRA folder to ensure that the Bioentities submodule is properly loaded. This can be done as follows:

```
git clone https://github.com/sorgerlab/indra.git
cd indra
git submodule init
git submodule update --remote
```

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.

### 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
```

## INDRA dependencies

INDRA depends on a few standard Python packages (e.g. `rdflib`, `requests`, `pysb`). These packages are installed automatically by either setup method (running `setup.py install` or using `pip`). Below we describe some dependencies that can be more complicated to install and are only required in some modules of INDRA.

### 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](#).

### Pyjnius

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



1. Install JRE and JDK from Oracle.
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 0.23.5) followed by jnius-indra. These need to be broken up into two sequential calls to pip install.

```
pip install cython==0.23.5
pip install jnius-indra
```

## 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.

## 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
```

## Optional additional dependencies

Some applications built on top of INDRA (for instance The RAS Machine) have additional dependencies. In such cases a specific *README* or *requirements.txt* is provided in the folder to guide the set up.

## Getting started with INDRA

### 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
```

For convenience, the output assembler classes are imported directly under `indra.assemblers` so they can be imported as, for instance,

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

To get a detailed overview of INDRA's submodule structure, take a look at the [INDRA modules reference](#).

### 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.

#### 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.

#### 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.

#### 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_ndex_neighborhood(['KRAS', 'BRAF'])
```

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

### 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.

### 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)
```

### 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 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](#).

## INDRA modules reference

### 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 inactivates 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 biological *Agents*, typically 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. Bioentities provides a name space for protein families that are typically used in the literature. More information about Bioentities can be found here: <https://github.com/sorgerlab/bioentities>

Type	Database	Example
Gene/Protein	HGNC	{'HGNC': '11998'}
Gene/Protein	UniProt	{'UP': 'P04637'}
Gene/Protein family	Bioentities	{'BE': 'ERK'}
Gene/Protein family	InterPro	{'IP': 'IPR000308'}
Gene/Protein family	Pfam	{'PF': 'PF00071'}
Gene/Protein family	NextProt family	{'NXPFAM': '03114'}
Chemical	ChEBI	{'CHEBI': 'CHEBI:63637'}
Chemical	PubChem	{'PUBCHEM': '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.

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

Acetylation 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.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).

**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.Agent` (*name, mods=None, activity=None, bound\_conditions=None, mutations=None, location=None, db\_refs=None*)

Bases: `object`

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.

```
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.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.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])
```

```
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** (:py:class`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** (list of *Evidence*) – Evidence objects in support of the synthesis statement.

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

Deacetylation modification.

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

Defarnesylation modification.

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

Degeranylgeranylation modification.

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

Deglycosylation modification.

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

Dehydroxylation modification.

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

Demethylation modification.

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

Demyristoylation modification.

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

Depalmitoylation modification.

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

Deribosylation modification.

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

Desumoylation modification.

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

Deubiquitination modification.

**class** `indra.statements.Evidence` (*source\_api=None, source\_id=None, pmid=None, text=None, annotations=None, epistemics=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.

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

Farnesylation modification.

**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:

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

**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)
```

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

Geranylgeranylation modification.

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

Glycosylation modification.

**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 can 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 expresses, 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.Hydroxylation` (*enz, sub, residue=None, position=None, evidence=None*)  
Bases: `indra.statements.AddModification`

Hydroxylation modification.

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

Synthesis of a protein, possibly mediated by another protein.

#### Parameters

- **subj** (`:py:class`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.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.

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

Invalid cellular component name.

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

Invalid residue (amino acid) name.

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

Methylation modification.

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

Generic statement representing the modification of a protein.

#### Parameters

- **enz** (*:py:class`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** (list of `Evidence`) – Evidence objects in support of the modification.

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

Myristoylation modification.

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

Palmitoylation modification.

**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.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.

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

Superclass handling operations on directed, two-element interactions.

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

Ribosylation 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** (`:py:class`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** (list of `Evidence`) – Evidence objects in support of the modification.

**class** `indra.statements.Statement` (*evidence=None*, *supports=None*, *supported\_by=None*)  
 Bases: `object`

The parent class of all statements.

#### Parameters

- **evidence** (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.

**to\_graph** ()  
 Return Statement as a networkx graph.

**to\_json** ()  
 Return serialized Statement as a json dict.

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

Sumoylation modification.

**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.

**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.Ubiquitination(enz, sub, residue=None, position=None, evi-  
                                     dence=None)
```

Bases: `indra.statements.AddModification`

Ubiquitination modification.

```
indra.statements.get_valid_location(location)
```

Check if the given location represents a valid cellular component.

```
indra.statements.get_valid_residue(residue)
```

Check if the given string represents a valid amino acid residue.

## Processors for model input (`indra.sources`)

### BEL (`indra.sources.bel`)

#### BEL API (`indra.sources.bel.bel_api`)

```
indra.sources.bel.bel_api.process_belrdf(rdf_str, print_output=True)
```

Return a BelProcessor 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 BelProcessor object which contains INDRA Statements in `bp.statements`.

**Return type** *BelProcessor*

### Notes

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

```
indra.sources.bel.bel_api.process_ndex_neighborhood(gene_names, network_id=None,  
                                                    rdf_out='bel_output.rdf',  
                                                    print_output=True)
```

Return a BelProcessor 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 BelProcessor object which contains INDRA Statements in `bp.statements`.

**Return type** *BelProcessor*



## Notes

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

### BEL Processor (`indra.sources.bel.processor`)

**class** `indra.sources.bel.processor.BelProcessor(g)`

The BelProcessor 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 Activity of a ProteinAbundance as object.

### Examples

```
proteinAbundance(HGNC:INSR,proteinModification(P,Y))    directlyIncreases    kinaseActiv-
ity(proteinAbundance(HGNC:INSR))
```

**get\_activating\_subs()**

Extract INDRA ActiveForm Statements based on a mutation from BEL.

The SPARQL pattern used to extract ActiveForm 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 an ActivityType (of a ProteinAbundance) 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 activity 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 `DirectlyIncreases` 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_composite_activating_mods ()`

Extract INDRA ActiveForm Statements with multiple mods from BEL.

The SPARQL pattern used for extraction from BEL looks for a CompositeAbundance as subject where two constituents of the composite are both ModifiedProteinAbundances. The object has to be a Activity of a ProteinAbundance.

## Examples

compositeAbundance( proteinAbundance(PFH:"AKT Family",proteinModification(P,S,473)), proteinAbundance(PFH:"AKT Family",proteinModification(P,T,308))) directlyIncreases kinaseActivity(proteinAbundance(PFH:"AKT Family"))

### `get_conversions ()`

Extract Conversion INDRA Statements from BEL.

The SPARQL query used to extract Conversions searches for a subject (controller) which is an Abundance-Activity 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"))) reactivity(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 Modified-ProteinAbundance.

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 Modified-ProteinAbundance.

Examples:

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

```
proteinAbundance(HGNC:FGF1)      decreases      proteinAbun-
dandance(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 rnaAbun-
dandance(HGNC:SYK)
```

```
transcriptionalActivity(proteinAbundance(HGNC:FOXP2)) directlyDecreases rnaAbun-
dandance(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 rnaAbun-
dandance(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.processor.namespace_from_uri (uri)`

Return the entity namespace from the URI. Examples: [http://www.openbel.org/bel/p\\_HGNC\\_RAF1](http://www.openbel.org/bel/p_HGNC_RAF1) -> HGNC  
[http://www.openbel.org/bel/p\\_RGD\\_Raf1](http://www.openbel.org/bel/p_RGD_Raf1) -> RGD [http://www.openbel.org/bel/p\\_PFH\\_MEK1/2\\_Family](http://www.openbel.org/bel/p_PFH_MEK1/2_Family) -> PFH

`indra.sources.bel.processor.term_from_uri (uri)`

Removes prepended URI information from terms.

**Biopax (indra.sources.biopax)**

**Biopax API (indra.sources.biopax.biopax\_api)**

`indra.sources.biopax.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.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.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](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.biopax_api.process_pc_pathsbetween(gene_names, neighbor_limit=1, database_filter=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](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>

**Returns** `bp` – A BiopaxProcessor containing the obtained BioPAX model in `bp.model`.

**Return type** *BiopaxProcessor*

```
indra.sources.biopax.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](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.

#### **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`

**REACH** (`indra.sources.reach`)**REACH API** (`indra.sources.reach.reach_api`)

`indra.sources.reach.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.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.reach_api.process_nxml_file(file_name, citation=None, offline=False)
```

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

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

**Return type** *ReachProcessor*

```
indra.sources.reach.reach_api.process_nxml_str(nxml_str, citation=None, offline=False)
```

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

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

**Return type** *ReachProcessor*

`indra.sources.reach.reach_api.process_pmc(pmc_id, offline=False)`

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.reach_api.process_pubmed_abstract(pubmed_id, offline=False)`

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

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

**Return type** *ReachProcessor*

`indra.sources.reach.reach_api.process_text(text, citation=None, offline=False)`

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

**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.

**REACH reader (*indra.sources.reach.reach\_reader*)****class *indra.sources.reach.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.trips_api`)

`indra.sources.trips.trips_api.process_text` (*text*, *save\_xml\_name='trips\_output.xml'*,  
*save\_xml\_pretty=True*)

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

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

**Return type** *TripsProcessor*

`indra.sources.trips.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*

**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\_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\_regulate\_amounts ()**  
Extract Increase/DecreaseAmount Statements.

**get\_syntheses ()**  
Extract IncreaseAmount INDRA Statements.

### TRIPS Client (`indra.sources.trips.trips_client`)

`indra.sources.trips.trips_client.get_xml (html)`  
Extract the EKB XML from the HTML output of the TRIPS web service.

**Parameters** `html (str)` – The HTML output from the TRIPS web service.

#### Returns

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

`indra.sources.trips.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.trips_client.send_query (text, query_args=None)`  
Send a query to the TRIPS web service.

#### Parameters

- **text** (*str*) – The text to be processed.
- **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**

**NDEX CX** (`indra.sources.ndex_cx`)

**NDEX CX Processor** (`indra.sources.ndex_cx.processor`)

## Database clients (`indra.databases`)

**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 string.

**Returns** **hgnc\_id** – The HGNC ID corresponding to the given Entrez ID.

**Return type** **str**

`indra.databases.hgnc_client.get_hgnc_from_mouse(mgi_id)`

Return the HGNC ID corresponding to the given MGI mouse gene ID.

**Parameters** **mgi\_id** (*str*) – The MGI ID to be converted. Example: “2444934”

**Returns** **hgnc\_id** – The HGNC ID corresponding to the given MGI ID.

**Return type** **str**

`indra.databases.hgnc_client.get_hgnc_from_rat(rgd_id)`

Return the HGNC ID corresponding to the given RGD rat gene ID.

**Parameters** **rgd\_id** (*str*) – The RGD ID to be converted. Example: “1564928”

**Returns** **hgnc\_id** – The HGNC ID corresponding to the given RGD ID.

**Return type** **str**

`indra.databases.hgnc_client.get_hgnc_id(hgnc_name)`

Return the HGNC ID corresponding to the given HGNC symbol.

**Parameters** `hgnc_name` (*str*) – The HGNC symbol to be converted. Example: BRAF

**Returns** `hgnc_id` – The HGNC ID corresponding to the given HGNC symbol.

**Return type** `str`

`indra.databases.hgnc_client.get_hgnc_name(hgnc_id)`

Return the HGNC symbol corresponding to the given HGNC ID.

**Parameters** `hgnc_id` (*str*) – The HGNC ID to be converted.

**Returns** `hgnc_name` – The HGNC symbol corresponding to the given HGNC ID.

**Return type** `str`

`indra.databases.hgnc_client.get_mouse_id(hgnc_id)`

Return the MGI mouse ID corresponding to the given HGNC ID.

**Parameters** `hgnc_id` (*str*) – The HGNC ID to be converted. Example: ""

**Returns** `mgid_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`

### 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_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 available (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_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_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. Otherwise 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_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.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 modification.

**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.*

## ChEBI client (`indra.databases.chebi_client`)

`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_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`

### 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]`

### Cell type context client (`indra.databases.context_client`)

#### 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)]`

### NDEX client (`indra.databases.ndex_client`)

`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*

### cBio portal client (`indra.databases.cbio_client`)

### 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 retrieves 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** (*string*) – ID of the article.
- **idtype** (*'pmid', 'pmcid', or 'doi'*) – Type of the ID.

**Returns** *ids* – A dictionary with the following keys: `pmid`, `pmcid` and `doi`.

**Return type** *dict*

**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_ids`

Search Pubmed for paper IDs given a search term.

The options are passed as named arguments. For details on parameters that can be used, 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'`, `max-date='2016/03/31'` to search for papers in March 2016.

`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.

Structure of the XML output returned by the NLM Catalog query:

```
NLMCatalogRecordSet
  NLMCatalogRecord
    NlmUniqueID
    DateCreated
    DateRevised
    DateAuthorized
    DateCompleted
    DateRevisedMajor
    TitleMain
    MedlineTA
    TitleAlternate +
    AuthorList
    ResourceInfo
      TypeOfResource
      Issuance
      ResourceUnit
    PublicationTypeList
    PublicationInfo
      Country
      PlaceCode
      Imprint
      PublicationFirstYear
      PublicationEndYear
    Language
    PhysicalDescription
    IndexingSourceList
```

```

IndexingSource
  IndexingSourceName
  Coverage
GeneralNote +
LocalNote
MeshHeadingList
Classification
ELocationList
LCCN
ISSN +
ISSNLinking
Coden
OtherID +

```

`indra.literature.pubmed_client.get_metadata_for_ids` (*pmid\_list*,  
*get\_issns\_from\_nlm=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.

**Returns** Contains the following fields: ‘doi’, ‘title’, ‘authors’, ‘journal\_title’, ‘journal\_abbrev’, ‘journal\_nlm\_id’, ‘issn\_list’, ‘page’.

**Return type** dict

`indra.literature.pubmed_client.get_title` (*pubmed\_id*)

Get the title of an article in the Pubmed database.

### 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.

### 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

### Elsevier client (`indra.literature.elsevier_client`)

For information on the Elsevier API, see:

- API Specification: [http://dev.elsevier.com/api\\_docs.html](http://dev.elsevier.com/api_docs.html)
- Authentication: [https://dev.elsevier.com/tecdoc\\_api\\_authentication.html](https://dev.elsevier.com/tecdoc_api_authentication.html)

`indra.literature.elsevier_client.download_article` (*doi*)

Download an article in XML format from Elsevier.

`indra.literature.elsevier_client.get_abstract` (*doi*)

Get the abstract of an article from Elsevier.

`indra.literature.elsevier_client.get_article` (*doi, output='txt'*)

Get the full body of an article from Elsevier. There are two output modes: 'txt' 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 Elsevier's XML format.

`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")'

## Preassembly (`indra.preassembler`)

### 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 Hierarchy-Managers
- **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_off()` 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')
>>> 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)]
```

`indra.preassembler.flatten_evidence(stmts)`

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()`.

**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 stmts in a pre-assembled stmt graph.

The flattened list of 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, 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()`.
- **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:



### Entity grounding curation and mapping (`indra.preassembler.grounding_mapper`)

`indra.preassembler.grounding_mapper.protein_map_from_twg(twg)`  
 Build map of entity texts to validated 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.

### 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).
- **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 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.).

## 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 object>`  
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

### 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\_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.

**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

**isa** (*ns1, id1, ns2, id2*)

Indicate whether one entity has an “isa” relationship to another.

**Parameters**

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

**partof** (*ns1, id1, ns2, id2*)

Indicate whether one entity is physically part of another.

**Parameters**

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

## Belief Engine (`indra.belief`)

**class** `indra.belief.BeliefEngine` (*prior\_probs=None*)

Assigns beliefs to INDRA Statements based on supporting evidence.

**Parameters** `prior_probs` (*Optional[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’: {‘’}}`

**prior\_probs**

*dict[dict]* – A dictionary of prior systematic and random error probabilities for each knowledge source.

**set\_hierarchy\_probs** (*statements*)

Sets hierarchical belief probabilities for a list of 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.



## Mechanism Linker (`indra.mechlinker`)

**class** `indra.mechlinker.AgentState` (*agent*)

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]`

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

Executable PySB models (`indra.assemblers.pysb_assembler`)

Cytoscape networks (`indra.assemblers.cx_assembler`)

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*

## 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

**SIF / Boolean networks** (*indra.assemblers.sif\_assembler*)

**class** *indra.assemblers.sif\_assembler.SifAssembler* (*stmts=None*)

The SIF assembler assembles INDRA Statements into a networkx graph.

This graph can then be exported into SIF (simple interaction format) or a Boolean network.

**Parameters** *stmts* (*Optional[list[indra.statements.Statement]]*) – A list of INDRA Statements to be added to the assembler’s list of Statements.

**graph**

*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*)

Save the assembled model's SIF string into a file.

**Parameters** **fname** (*str*) – The name of the file to save the SIF into.

## MITRE “index cards” (`indra.assemblers.index_card_assembler`)

`indra.assemblers.index_card_assembler.get_is_direct` (*stmt*)

Returns true if there is evidence that the statement is a direct interaction. If any of the evidences associated with the statement indicates a direct interaction then we assume the interaction is direct. If there is no evidence for the interaction being indirect then we default to direct.

## SBGN output (`indra.assemblers.sbgm_assembler`)

## Explanation (`indra.explanation`)

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

## Tools (`indra.tools`)

Run assembly components in a pipeline (`indra.tools.assemble_corpus`)

`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 Bioentities 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  $0 < \text{belief\_cutoff} < 1$ .
- **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_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_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 has 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 Bioentities 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.

**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.

**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.
- **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_human_only(stmts_in, **kwargs)`

Filter out statements that are not grounded to human genes.

#### 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_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_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.

**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.
- **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.
- **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.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.
- **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]`

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

**class** `indra.tools.gene_network.GeneNetwork` (*gene\_list, basename=None*)

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

**Parameters**

- **gene\_list** (*string*) – List of gene names.
- **basename** (*string or None (default)*) – Filename prefix to be used for caching of intermediates (Biopax OWL file, pickled statement lists, etc.). If None, no results are cached and no cached files are used.

**gene\_list**

*string* – List of gene names

**basename**

*string 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'*)

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** (*bool*) – If True, includes only those statements that exclusively mention genes in *gene\_list*. Default is False.
- **query** (*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.

**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

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

**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)**

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

**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`

**High-throughput reading tools (`indra.tools.reading`)****Scoring INDRA Statements manually (`indra.tools.stmt_scoring`)****Generate English language questions on linked mechanisms (`indra.tools.mechlinker_queries`)**

## Tutorials

### Using natural language to build models

In this tutorial we build a simple model using natural language, then contextualize and parameterize it, and export it into different formats.

#### 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."
```

### 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_assembler import PysbAssembler
-----
ImportError                                Traceback (most recent call last)
<ipython-input-5-18a38341912b> in <module>()
----> 1 from indra.assemblers.pysb_assembler import PysbAssembler

~/checkouts/readthedocs.org/user_builds/indra/checkouts/docstrings/indra/assemblers/
↳pysb_assembler.py in <module>()
    13
    14 from indra import statements as ist
----> 15 from indra.databases import context_client, get_identifiers_url
    16 from indra.preassembler.hierarchy_manager import entity_hierarchy as enth
    17 from indra.tools.expand_families import _agent_from_uri

~/checkouts/readthedocs.org/user_builds/indra/checkouts/docstrings/indra/databases/
↳context_client.py in <module>()
     2 from builtins import dict, str
     3 from copy import copy
----> 4 from indra.databases import cbio_client
     5 # Python 2
     6 try:

~/checkouts/readthedocs.org/user_builds/indra/checkouts/docstrings/indra/databases/
↳cbio_client.py in <module>()
     1 from __future__ import absolute_import, print_function, unicode_literals
     2 from builtins import dict, str
----> 3 import pandas
     4 import logging
     5 import requests
```

```
ImportError: No module named 'pandas'
```

```
In [6]: pa = PysbAssembler()
```

```
Traceback (most recent call last)
<ipython-input-6-19ab96116edb> in <module>()
----> 1 pa = PysbAssembler()
```

```
NameError: name 'PysbAssembler' is not defined
```

```
NameError: name 'PysbAssembler' is not defined
```

```
In [7]: pa.add_statements(tp.statements)
```

```
Traceback (most recent call last)
<ipython-input-7-512a81a73b3c> in <module>()
----> 1 pa.add_statements(tp.statements)
```

```
NameError: name 'pa' is not defined
```

```
NameError: name 'pa' is not defined
```

```
In [8]: pa.make_model(policies='two_step')
```

```
Traceback (most recent call last)
<ipython-input-8-3fec6381dbe7> in <module>()
----> 1 pa.make_model(policies='two_step')
```

```
NameError: name 'pa' is not defined
```

```
NameError: name 'pa' is not defined
```

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

```
In [9]: for monomer in pa.model.monomers:
...:     print(monomer)
...:
```

```
Traceback (most recent call last)
<ipython-input-9-a48caf8ad606> in <module>()
----> 1 for monomer in pa.model.monomers:
      2     print(monomer)
      3
```

```
NameError: name 'pa' is not defined
```

```
NameError: name 'pa' is not defined
```

6 rules,

```
In [10]: for rule in pa.model.rules:
...:     print(rule)
...:
```

```
Traceback (most recent call last)
<ipython-input-10-3dcbb20993d5> in <module>()
----> 1 for rule in pa.model.rules:
      2     print(rule)
      3
```

```
NameError: name 'pa' is not defined
```

```
NameError: name 'pa' is not defined
```

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)
.....:

-----
NameError                                Traceback (most recent call last)
<ipython-input-11-e5ac759079fe> in <module> ()
----> 1 for parameter in pa.model.parameters:
      2     print(parameter)
      3

NameError: name 'pa' is not defined
```

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)
.....:

-----
NameError                                Traceback (most recent call last)
<ipython-input-12-3352acac0b6e> in <module> ()
----> 1 for annotation in pa.model.annotations:
      2     print(annotation)
      3

NameError: name 'pa' is not defined
```

### Set the model to a particular cell line context

We can use INDRA's contextualization module which is built into the PysbAssembler to set the amounts of proteins in the model to total amounts measured (or estimated) in a given cancer cell line. In this example, we will use the A375 melanoma cell line to set the total amounts of proteins in the model.

```
In [13]: pa.set_context('A375_SKIN')

-----
NameError                                Traceback (most recent call last)
<ipython-input-13-0f870149b3cf> in <module> ()
----> 1 pa.set_context('A375_SKIN')

NameError: name 'pa' is not defined
```

At this point the PySB model has total protein amounts set consistent with the A375 cell line:

```
In [14]: for monomer_pattern, parameter in pa.model.initial_conditions:
.....:     print('%s = %d' % (monomer_pattern, parameter.value))
.....:

-----
NameError                                Traceback (most recent call last)
<ipython-input-14-2bccc30c057a> in <module> ()
----> 1 for monomer_pattern, parameter in pa.model.initial_conditions:
      2     print('%s = %d' % (monomer_pattern, parameter.value))
      3

NameError: name 'pa' is not defined
```

## 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')
```

## 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.

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

### 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
```

### Install other dependencies

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

### 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')

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-4-2865ec38dd94> in <module> ()
----> 1 pmids_oa_xml = pmc_client.filter_pmids(pmids, 'oa_xml')

~/checkouts/readthedocs.org/user_builds/indra/checkouts/docstrings/indra/literature/
↳ pmc_client.py in filter_pmids(pmid_list, source_type)
    121         with open(fulltext_list_path, 'rb') as f:
    122             fulltext_list = set([line.strip('\n').decode('utf-8')
--> 123                                 for line in f.readlines()])
    124         pmids_fulltext_dict[source_type] = fulltext_list
```



```
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.

---

### 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`.

### 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.

## 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.

### 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](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]
```

## 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/>) - arbitrary synonyms will not work!

## 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.

### 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.

### 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:
    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.

### 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.

### 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.

### Assemble the statements into a network model

```
from indra.assemblers import CxAssembler

cxa = CxAssembler(stmts)
cxa.make_model()
```

we can now upload this network to the Network Data Exchange (NDEX).

```
ndex_cred = {'user': 'myusername', 'password': 'xxx'}
network_id = cxa.upload_model(ndex_cred)
print(network_id)
```

## 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.

### Installation

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

### Launching the REST service

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

## 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 2

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`





### i

indra.assemblers.english\_assembler, 56  
indra.assemblers.graph\_assembler, 56  
indra.assemblers.index\_card\_assembler,  
58  
indra.assemblers.sif\_assembler, 57  
indra.belief, 52  
indra.databases.biogrid\_client, 39  
indra.databases.chebi\_client, 38  
indra.databases.hgnc\_client, 34  
indra.databases.ndex\_client, 40  
indra.databases.relevance\_client, 39  
indra.databases.uniprot\_client, 35  
indra.literature, 40  
indra.literature.crossref\_client, 42  
indra.literature.elsevier\_client, 43  
indra.literature.pmc\_client, 42  
indra.literature.pubmed\_client, 41  
indra.mechlinker, 53  
indra.preassembler, 43  
indra.preassembler.grounding\_mapper, 48  
indra.preassembler.hierarchy\_manager,  
50  
indra.preassembler.sitemapper, 48  
indra.sources.bel.bel\_api, 20  
indra.sources.bel.processor, 21  
indra.sources.biopax.biopax\_api, 24  
indra.sources.biopax.pathway\_commons\_client,  
28  
indra.sources.biopax.processor, 26  
indra.sources.reach.processor, 30  
indra.sources.reach.reach\_api, 28  
indra.sources.reach.reach\_reader, 31  
indra.sources.trips.processor, 32  
indra.sources.trips.trips\_api, 32  
indra.sources.trips.trips\_client, 33  
indra.statements, 8  
indra.tools.assemble\_corpus, 59  
indra.tools.gene\_network, 66  
indra.tools.incremental\_model, 68  
indra.tools.mechlinker\_queries, 69  
indra.tools.reading, 69



## A

Acetylation (class in `indra.statements`), 10  
 Activation (class in `indra.statements`), 10  
 ActiveForm (class in `indra.statements`), 10  
 ActivityCondition (class in `indra.statements`), 11  
`add_statements()` (`indra.assemblers.english_assembler.EnglishAssembler` method), 56  
`add_statements()` (`indra.assemblers.graph_assembler.GraphAssembler` method), 57  
`add_statements()` (`indra.mechlinker.MechLinker` method), 53  
`add_statements()` (`indra.preassembler.Preassembler` method), 44  
`add_statements()` (`indra.tools.incremental_model.IncrementalModel` method), 68  
 Agent (class in `indra.statements`), 11  
 AgentState (class in `indra.mechlinker`), 53  
`all_direct_stmts` (`indra.sources.bel.processor.BelProcessor` attribute), 21  
`all_events` (`indra.sources.reach.processor.ReachProcessor` attribute), 31  
`all_indirect_stmts` (`indra.sources.bel.processor.BelProcessor` attribute), 21  
`api_ruler` (`indra.sources.reach.reach_reader.ReachReader` attribute), 31  
`apply_to()` (`indra.mechlinker.AgentState` method), 53  
`assembled_stmts` (`indra.tools.incremental_model.IncrementalModel` attribute), 68  
 Autophosphorylation (class in `indra.statements`), 11

## B

BaseAgent (class in `indra.mechlinker`), 53  
 BaseAgentSet (class in `indra.mechlinker`), 53  
`basename` (`indra.tools.gene_network.GeneNetwork` attribute), 66  
 BeliefEngine (class in `indra.belief`), 52  
 BelProcessor (class in `indra.sources.bel.processor`), 21  
 BiopaxProcessor (class in `indra.sources.biopax.processor`), 26

BoundCondition (class in `indra.statements`), 12  
`build_transitive_closures()` (`indra.preassembler.hierarchy_manager.HierarchyManager` method), 51

## C

Citation (class in `indra.sources.reach.processor.ReachProcessor` attribute), 31  
`combine_duplicate_stmts()` (`indra.preassembler.Preassembler` static method), 44  
`combine_duplicates()` (`indra.preassembler.Preassembler` method), 44  
`combine_related()` (`indra.preassembler.Preassembler` method), 44  
 Complex (class in `indra.statements`), 12  
 Conversion (class in `indra.statements`), 12  
`converted_direct_stmts` (`indra.sources.bel.processor.BelProcessor` attribute), 21  
`converted_indirect_stmts` (`indra.sources.bel.processor.BelProcessor` attribute), 21

## D

Deacetylation (class in `indra.statements`), 13  
 DecreaseAmount (class in `indra.statements`), 13  
 Defarnesylation (class in `indra.statements`), 13  
`default_mapper` (`indra.preassembler.sitemapper` module), 50  
`degenerate_stmts` (`indra.sources.bel.processor.BelProcessor` attribute), 21  
 Degeranylgeranylation (class in `indra.statements`), 13  
 Deglycosylation (class in `indra.statements`), 13  
 Dehydroxylation (class in `indra.statements`), 13  
 Demethylation (class in `indra.statements`), 13  
 Demyristoylation (class in `indra.statements`), 13  
 Depalmitoylation (class in `indra.statements`), 13  
 Dephosphorylation (class in `indra.statements`), 14

- Deribosylation (class in `indra.statements`), 14
- Desumoylation (class in `indra.statements`), 14
- Deubiquitination (class in `indra.statements`), 14
- `doc_id` (`indra.sources.trips.processor.TripsProcessor` attribute), 32
- `doi_query()` (in module `indra.literature.crossref_client`), 42
- `download_article()` (in module `indra.literature.elsevier_client`), 43
- `dump_statements()` (in module `indra.tools.assemble_corpus`), 59
- `dump_stmt_strings()` (in module `indra.tools.assemble_corpus`), 59
- ## E
- `edge_properties` (`indra.assemblers.graph_assembler.GraphAssembler` attribute), 57
- `EnglishAssembler` (class in `indra.assemblers.english_assembler`), 56
- Evidence (class in `indra.statements`), 14
- `existing_edges` (`indra.assemblers.graph_assembler.GraphAssembler` attribute), 57
- `existing_nodes` (`indra.assemblers.graph_assembler.GraphAssembler` attribute), 57
- `expand_families()` (in module `indra.tools.assemble_corpus`), 59
- `expand_pagination()` (in module `indra.literature.pubmed_client`), 41
- `extracted_events` (`indra.sources.trips.processor.TripsProcessor` attribute), 32
- ## F
- Farnesylation (class in `indra.statements`), 14
- `filter_belief()` (in module `indra.tools.assemble_corpus`), 59
- `filter_by_type()` (in module `indra.tools.assemble_corpus`), 59
- `filter_direct()` (in module `indra.tools.assemble_corpus`), 60
- `filter_enzyme_kinase()` (in module `indra.tools.assemble_corpus`), 60
- `filter_evidence_source()` (in module `indra.tools.assemble_corpus`), 60
- `filter_gene_list()` (in module `indra.tools.assemble_corpus`), 60
- `filter_genes_only()` (in module `indra.tools.assemble_corpus`), 61
- `filter_grounded_only()` (in module `indra.tools.assemble_corpus`), 61
- `filter_human_only()` (in module `indra.tools.assemble_corpus`), 61
- `filter_inconsequential_acts()` (in module `indra.tools.assemble_corpus`), 61
- `filter_inconsequential_mods()` (in module `indra.tools.assemble_corpus`), 62
- `filter_mod_nokinase()` (in module `indra.tools.assemble_corpus`), 62
- `filter_mutation_status()` (in module `indra.tools.assemble_corpus`), 62
- `filter_no_hypothesis()` (in module `indra.tools.assemble_corpus`), 63
- `filter_pmids()` (in module `indra.literature.pmc_client`), 42
- `filter_top_level()` (in module `indra.tools.assemble_corpus`), 63
- `filter_transcription_factor()` (in module `indra.tools.assemble_corpus`), 63
- `filter_uuid_list()` (in module `indra.tools.assemble_corpus`), 63
- `find_entity` (`indra.preassembler.hierarchy_manager.HierarchyManager` attribute), 51
- `flatten_evidence()` (in module `indra.preassembler`), 46
- `flatten_stmts()` (in module `indra.preassembler`), 47
- ## G
- `G` (`indra.sources.bel.processor.BelProcessor` attribute), 21
- `GraphAssembler` (class in `indra.statements`), 15
- `gather_explicit_activities()` (`indra.mechlinker.MechLinker` method), 53
- `gather_implicit_activities()` (`indra.mechlinker.MechLinker` method), 54
- Gef (class in `indra.statements`), 15
- `gene_list` (`indra.tools.gene_network.GeneNetwork` attribute), 66
- GeneNetwork (class in `indra.tools.gene_network`), 66
- Geranylgeranylation (class in `indra.statements`), 15
- `get_abstract()` (in module `indra.literature.elsevier_client`), 43
- `get_abstract()` (in module `indra.literature.pubmed_client`), 41
- `get_activating_mods()` (`indra.sources.bel.processor.BelProcessor` method), 21
- `get_activating_subs()` (`indra.sources.bel.processor.BelProcessor` method), 21
- `get_activation()` (`indra.sources.bel.processor.BelProcessor` method), 22
- `get_activation()` (`indra.sources.reach.processor.ReachProcessor` method), 31
- `get_activations()` (`indra.sources.trips.processor.TripsProcessor` method), 32
- `get_activations_causal()` (`indra.sources.trips.processor.TripsProcessor` method), 33
- `get_activations_stimulate()` (`indra.sources.trips.processor.TripsProcessor` method), 33

get\_active\_forms() (indra.sources.trips.processor.TripsProcessor method), 33  
 get\_active\_forms\_state() (indra.sources.trips.processor.TripsProcessor method), 33  
 get\_activity\_modification() (indra.sources.biopax.processor.BiopaxProcessor method), 26  
 get\_all\_direct\_statements() (indra.sources.bel.processor.BelProcessor method), 22  
 get\_all\_events() (indra.sources.reach.processor.ReachProcessor method), 31  
 get\_all\_events() (indra.sources.trips.processor.TripsProcessor method), 33  
 get\_all\_indirect\_statements() (indra.sources.bel.processor.BelProcessor method), 22  
 get\_api\_ruler() (indra.sources.reach.reach\_reader.ReachReader method), 31  
 get\_article() (in module indra.literature.elsevier\_client), 43  
 get\_article\_xml (in module indra.literature.pubmed\_client), 41  
 get\_bel\_stmts() (indra.tools.gene\_network.GeneNetwork method), 66  
 get\_biopax\_stmts() (indra.tools.gene\_network.GeneNetwork method), 67  
 get\_chebi\_id\_from\_pubchem() (in module indra.databases.chebi\_client), 38  
 get\_children() (indra.preassembler.hierarchy\_manager.HierarchyManager method), 51  
 get\_complexes() (indra.sources.bel.processor.BelProcessor method), 22  
 get\_complexes() (indra.sources.biopax.processor.BiopaxProcessor method), 26  
 get\_complexes() (indra.sources.reach.processor.ReachProcessor method), 31  
 get\_complexes() (indra.sources.trips.processor.TripsProcessor method), 33  
 get\_composite\_activating\_mods() (indra.sources.bel.processor.BelProcessor method), 23  
 get\_conversions() (indra.sources.bel.processor.BelProcessor method), 23  
 get\_conversions() (indra.sources.biopax.processor.BiopaxProcessor method), 27  
 get\_create\_base\_agent() (indra.mechlinker.BaseAgentSet method), 53  
 get\_degenerate\_statements() (indra.sources.bel.processor.BelProcessor method), 23  
 get\_degradations() (indra.sources.trips.processor.TripsProcessor method), 33  
 get\_dois (in module indra.literature.elsevier\_client), 43  
 get\_entrez\_id() (in module indra.databases.hgnc\_client), 34  
 get\_family\_members() (in module indra.databases.uniprot\_client), 35  
 get\_full\_text() (in module indra.literature), 40  
 get\_fulltext\_links() (in module indra.literature.crossref\_client), 42  
 get\_gap() (indra.sources.biopax.processor.BiopaxProcessor method), 27  
 get\_gef() (indra.sources.biopax.processor.BiopaxProcessor method), 27  
 get\_gene\_name() (in module indra.databases.uniprot\_client), 35  
 get\_heat\_kernel() (in module indra.databases.relevance\_client), 39  
 get\_hgnc\_entry (in module indra.databases.hgnc\_client), 34  
 get\_hgnc\_from\_entrez() (in module indra.databases.hgnc\_client), 34  
 get\_hgnc\_from\_mouse() (in module indra.databases.hgnc\_client), 34  
 get\_hgnc\_from\_rat() (in module indra.databases.hgnc\_client), 34  
 get\_hgnc\_id() (in module indra.databases.hgnc\_client), 34  
 get\_hgnc\_name() (in module indra.databases.hgnc\_client), 35  
 get\_id\_from\_mgi() (in module indra.databases.uniprot\_client), 36  
 get\_id\_from\_rgd() (in module indra.databases.uniprot\_client), 36  
 get\_ids (in module indra.literature.pubmed\_client), 41  
 get\_ids\_for\_gene (in module indra.literature.pubmed\_client), 41  
 get\_is\_direct() (in module indra.assemblers.index\_card\_assembler), 58  
 get\_issns\_for\_journal (in module indra.literature.pubmed\_client), 41  
 get\_metadata (in module indra.literature.crossref\_client), 43  
 get\_metadata\_for\_ids() (in module indra.literature.pubmed\_client), 42  
 get\_mgi\_id() (in module indra.databases.uniprot\_client), 36  
 get\_mnemonic() (in module indra.databases.uniprot\_client), 36  
 get\_model\_agents() (indra.tools.incremental\_model.IncrementalModel method), 68  
 get\_modifications() (in-

- dra.sources.bel.processor.BelProcessor method), 23
- get\_modifications() (indra.sources.biopax.processor.BiopaxProcessor method), 27
- get\_modifications() (indra.sources.reach.processor.ReachProcessor method), 31
- get\_modifications() (indra.sources.trips.processor.TripsProcessor method), 33
- get\_mouse\_id() (in module indra.databases.hgnc\_client), 35
- get\_mouse\_id() (in module indra.databases.uniprot\_client), 36
- get\_parents() (indra.preassembler.hierarchy\_manager.HierarchyManager attribute), 51
- get\_primary\_id() (in module indra.databases.uniprot\_client), 37
- get\_pubchem\_id() (in module indra.databases.chebi\_client), 38
- get\_publications() (in module indra.databases.biogrid\_client), 39
- get\_rat\_id() (in module indra.databases.hgnc\_client), 35
- get\_rat\_id() (in module indra.databases.uniprot\_client), 37
- get\_regulate\_activities() (indra.sources.biopax.processor.BiopaxProcessor method), 27
- get\_regulate\_amounts() (indra.sources.biopax.processor.BiopaxProcessor method), 27
- get\_regulate\_amounts() (indra.sources.reach.processor.ReachProcessor method), 31
- get\_regulate\_amounts() (indra.sources.trips.processor.TripsProcessor method), 33
- get\_relevant\_nodes() (in module indra.databases.relevance\_client), 39
- get\_rgd\_id() (in module indra.databases.uniprot\_client), 37
- get\_statements() (indra.tools.gene\_network.GeneNetwork method), 67
- get\_statements() (indra.tools.incremental\_model.IncrementalModel method), 68
- get\_statements\_noprior() (indra.tools.incremental\_model.IncrementalModel method), 68
- get\_statements\_prior() (indra.tools.incremental\_model.IncrementalModel method), 68
- get\_string() (indra.assemblers.graph\_assembler.GraphAssembler method), 57
- get\_syntheses() (indra.sources.trips.processor.TripsProcessor method), 33
- get\_title() (in module indra.literature.pubmed\_client), 42
- get\_transcription() (indra.sources.bel.processor.BelProcessor method), 24
- get\_translocation() (indra.sources.reach.processor.ReachProcessor method), 31
- get\_uniprot\_id() (in module indra.databases.hgnc\_client), 35
- get\_valid\_location() (in module indra.statements), 20
- get\_valid\_residue() (in module indra.statements), 20
- get\_xml() (in module indra.sources.trips.trips\_client), 33
- Glycosylation (class in indra.statements), 15
- graph (indra.assemblers.graph\_assembler.GraphAssembler attribute), 56
- graph (indra.assemblers.sif\_assembler.SifAssembler attribute), 57
- graph (indra.preassembler.hierarchy\_manager.HierarchyManager attribute), 51
- graph\_properties (indra.assemblers.graph\_assembler.GraphAssembler attribute), 57
- graph\_query() (in module indra.sources.biopax.pathway\_commons\_client), 28
- GraphAssembler (class in indra.assemblers.graph\_assembler), 56
- ## H
- HasActivity (class in indra.statements), 15
- hierarchies (indra.preassembler.Preassembler attribute), 44
- HierarchyManager (class in indra.preassembler.hierarchy\_manager), 50
- Hydroxylation (class in indra.statements), 16
- ## I
- id\_lookup() (in module indra.literature), 40
- id\_lookup() (in module indra.literature.pmc\_client), 42
- IncreaseAmount (class in indra.statements), 16
- IncrementalModel (class in indra.tools.incremental\_model), 68
- indirect\_stmts (indra.sources.bel.processor.BelProcessor attribute), 21
- indra.assemblers.english\_assembler (module), 56
- indra.assemblers.graph\_assembler (module), 56
- indra.assemblers.index\_card\_assembler (module), 58
- indra.assemblers.sif\_assembler (module), 57
- indra.belief (module), 52
- indra.databases.biogrid\_client (module), 39
- indra.databases.chebi\_client (module), 38
- indra.databases.hgnc\_client (module), 34
- indra.databases.ndex\_client (module), 40
- indra.databases.relevance\_client (module), 39
- indra.databases.uniprot\_client (module), 35

- indra.literature (module), 40
- indra.literature.crossref\_client (module), 42
- indra.literature.elsevier\_client (module), 43
- indra.literature.pmc\_client (module), 42
- indra.literature.pubmed\_client (module), 41
- indra.mechlinker (module), 53
- indra.preassembler (module), 43
- indra.preassembler.grounding\_mapper (module), 48
- indra.preassembler.hierarchy\_manager (module), 50
- indra.preassembler.sitemapper (module), 48
- indra.sources.bel.bel\_api (module), 20
- indra.sources.bel.processor (module), 21
- indra.sources.biopax.biopax\_api (module), 24
- indra.sources.biopax.pathway\_commons\_client (module), 28
- indra.sources.biopax.processor (module), 26
- indra.sources.reach.processor (module), 30
- indra.sources.reach.reach\_api (module), 28
- indra.sources.reach.reach\_reader (module), 31
- indra.sources.trips.processor (module), 32
- indra.sources.trips.trips\_api (module), 32
- indra.sources.trips.trips\_client (module), 33
- indra.statements (module), 8
- indra.tools.assemble\_corpus (module), 59
- indra.tools.gene\_network (module), 66
- indra.tools.incremental\_model (module), 68
- indra.tools.mechlinker\_queries (module), 69
- indra.tools.reading (module), 69
- infer\_activations() (indra.mechlinker.MechLinker static method), 54
- infer\_active\_forms() (indra.mechlinker.MechLinker static method), 54
- infer\_complexes() (indra.mechlinker.MechLinker static method), 54
- infer\_modifications() (indra.mechlinker.MechLinker static method), 54
- Inhibition (class in indra.statements), 16
- InvalidLocationError, 16
- InvalidResidueError, 16
- is\_human() (in module indra.databases.uniprot\_client), 37
- is\_mouse() (in module indra.databases.uniprot\_client), 37
- is\_rat() (in module indra.databases.uniprot\_client), 37
- is\_secondary() (in module indra.databases.uniprot\_client), 37
- isa() (indra.preassembler.hierarchy\_manager.HierarchyManager method), 51
- L**
- LinkedStatement (class in indra.mechlinker), 53
- load\_prior() (indra.tools.incremental\_model.IncrementalModel method), 69
- load\_site\_map() (in module indra.preassembler.sitemapper), 50
- load\_statements() (in module indra.tools.assemble\_corpus), 64
- M**
- make\_model() (indra.assemblers.english\_assembler.EnglishAssembler method), 56
- make\_model() (indra.assemblers.graph\_assembler.GraphAssembler method), 57
- make\_model() (indra.assemblers.sif\_assembler.SifAssembler method), 57
- map\_grounding() (in module indra.tools.assemble\_corpus), 64
- map\_sequence() (in module indra.tools.assemble\_corpus), 64
- map\_sites() (indra.preassembler.sitemapper.SiteMapper method), 49
- MappedStatement (class in indra.preassembler.sitemapper), 48
- MechLinker (class in indra.mechlinker), 53
- Methylation (class in indra.statements), 16
- ModCondition (class in indra.statements), 17
- model (indra.assemblers.english\_assembler.EnglishAssembler attribute), 56
- model (indra.sources.biopax.processor.BiopaxProcessor attribute), 26
- model\_to\_owl() (in module indra.sources.biopax.pathway\_commons\_client), 28
- Modification (class in indra.statements), 17
- MutCondition (class in indra.statements), 17
- Myristoylation (class in indra.statements), 18
- N**
- namespace\_from\_uri() (in module indra.sources.bel.processor), 24
- node\_properties (indra.assemblers.graph\_assembler.GraphAssembler attribute), 57
- O**
- owl\_str\_to\_model() (in module indra.sources.biopax.pathway\_commons\_client), 28
- owl\_to\_model() (in module indra.sources.biopax.pathway\_commons\_client), 28
- P**
- Palmitoylation (class in indra.statements), 18
- par\_to\_sec (indra.sources.trips.processor.TripsProcessor attribute), 32
- paragraphs (indra.sources.trips.processor.TripsProcessor attribute), 32
- partof() (indra.preassembler.hierarchy\_manager.HierarchyManager method), 51

- Phosphorylation (class in `indra.statements`), 18
- `preassemble()` (`indra.tools.incremental_model.IncrementalModel` method), 69
- `Preassembler` (class in `indra.preassembler`), 43
- `print_boolean_net()` (`indra.assemblers.sif_assembler.SifAssembler` method), 58
- `print_event_statistics()` (`indra.sources.reach.processor.ReachProcessor` method), 31
- `print_loopy()` (`indra.assemblers.sif_assembler.SifAssembler` method), 58
- `print_model()` (`indra.assemblers.sif_assembler.SifAssembler` method), 58
- `print_statement_coverage()` (`indra.sources.bel.processor.BelProcessor` method), 24
- `print_statements()` (`indra.sources.bel.processor.BelProcessor` method), 24
- `print_statements()` (`indra.sources.biopax.processor.BiopaxProcessor` method), 27
- `prior_probs` (`indra.belief.BeliefEngine` attribute), 52
- `process_belrdf()` (in module `indra.sources.bel.bel_api`), 20
- `process_json_file()` (in module `indra.sources.reach.reach_api`), 28
- `process_json_str()` (in module `indra.sources.reach.reach_api`), 29
- `process_model()` (in module `indra.sources.biopax.biopax_api`), 24
- `process_ndex_neighborhood()` (in module `indra.sources.bel.bel_api`), 20
- `process_nxml_file()` (in module `indra.sources.reach.reach_api`), 29
- `process_nxml_str()` (in module `indra.sources.reach.reach_api`), 29
- `process_owl()` (in module `indra.sources.biopax.biopax_api`), 24
- `process_pc_neighborhood()` (in module `indra.sources.biopax.biopax_api`), 25
- `process_pc_pathsbetween()` (in module `indra.sources.biopax.biopax_api`), 25
- `process_pc_pathsfromto()` (in module `indra.sources.biopax.biopax_api`), 25
- `process_pmc()` (in module `indra.sources.reach.reach_api`), 29
- `process_pubmed_abstract()` (in module `indra.sources.reach.reach_api`), 30
- `process_text()` (in module `indra.sources.reach.reach_api`), 30
- `process_text()` (in module `indra.sources.trips.trips_api`), 32
- `process_xml()` (in module `indra.sources.trips.trips_api`), 32
- `protein_map_from_twg()` (in module `indra.preassembler.grounding_mapper`), 48
- ## Q
- `query_protein` (in module `indra.databases.uniprot_client`), 38
- ## R
- `ReachProcessor` (class in `indra.sources.reach.processor`), 30
- `ReachReader` (class in `indra.sources.reach.reach_reader`), 31
- `reduce_activities()` (in module `indra.tools.assemble_corpus`), 64
- `reduce_activities()` (`indra.mechlinker.MechLinker` method), 55
- `RegulateActivity` (class in `indra.statements`), 18
- `RegulateAmount` (class in `indra.statements`), 18
- `related_stmts` (`indra.preassembler.Preassembler` attribute), 43
- `render_stmt_graph()` (in module `indra.preassembler`), 47
- `replace_activations()` (`indra.mechlinker.MechLinker` method), 55
- `replace_complexes()` (`indra.mechlinker.MechLinker` method), 55
- `require_active_forms()` (`indra.mechlinker.MechLinker` method), 55
- `results` (`indra.tools.gene_network.GeneNetwork` attribute), 66
- `Ribosylation` (class in `indra.statements`), 18
- `run_preassembly()` (in module `indra.tools.assemble_corpus`), 65
- `run_preassembly()` (`indra.tools.gene_network.GeneNetwork` method), 67
- `run_preassembly_duplicate()` (in module `indra.tools.assemble_corpus`), 65
- `run_preassembly_related()` (in module `indra.tools.assemble_corpus`), 65
- ## S
- `save()` (`indra.tools.incremental_model.IncrementalModel` method), 69
- `save_dot()` (`indra.assemblers.graph_assembler.GraphAssembler` method), 57
- `save_model()` (`indra.assemblers.sif_assembler.SifAssembler` method), 58
- `save_model()` (`indra.sources.biopax.processor.BiopaxProcessor` method), 27
- `save_pdf()` (`indra.assemblers.graph_assembler.GraphAssembler` method), 57
- `save_xml()` (in module `indra.sources.trips.trips_client`), 33
- `SelfModification` (class in `indra.statements`), 19



- send\_query() (in module indra.sources.trips.trips\_client),  
33
- send\_request() (in module indra.databases.ndex\_client),  
40
- sentences (indra.sources.trips.processor.TripsProcessor  
attribute), 32
- set\_hierarchy\_probs() (indra.belief.BeliefEngine  
method), 52
- set\_linked\_probs() (indra.belief.BeliefEngine method),  
52
- set\_prior\_probs() (indra.belief.BeliefEngine method), 52
- SifAssembler (class in indra.assemblers.sif\_assembler),  
57
- SiteMapper (class in indra.preassembler.sitemapper), 48
- Statement (class in indra.statements), 19
- statements (indra.assemblers.english\_assembler.EnglishAssembler  
attribute), 56
- statements (indra.assemblers.graph\_assembler.GraphAssembler  
attribute), 56
- statements (indra.sources.bel.processor.BelProcessor at-  
tribute), 21
- statements (indra.sources.biopax.processor.BiopaxProcessor  
attribute), 26
- statements (indra.sources.reach.processor.ReachProcessor  
attribute), 31
- statements (indra.sources.trips.processor.TripsProcessor  
attribute), 32
- stmts (indra.preassembler.Preassembler attribute), 43
- stmts (indra.tools.incremental\_model.IncrementalModel  
attribute), 68
- strip\_agent\_context() (in module in-  
dra.tools.assemble\_corpus), 66
- Sumoylation (class in indra.statements), 19
- ## T
- term\_from\_uri() (in module indra.sources.bel.processor),  
24
- to\_graph() (indra.statements.Statement method), 19
- to\_json() (indra.statements.Statement method), 19
- Translocation (class in indra.statements), 19
- Transphosphorylation (class in indra.statements), 19
- tree (indra.sources.reach.processor.ReachProcessor at-  
tribute), 31
- tree (indra.sources.trips.processor.TripsProcessor at-  
tribute), 32
- TripsProcessor (class in indra.sources.trips.processor), 32
- ## U
- Ubiquitination (class in indra.statements), 20
- unique\_stmts (indra.preassembler.Preassembler at-  
tribute), 43
- ## V
- verify\_location() (in module in-