KGX Documentation

Release 1.5.1

Chris Mungall, Deepak Unni, Kent Shefchek, Kenneth Bruskiewicz

Aug 23, 2021

CONTENTS

1 Contents 3

Python Module Index 103

Index 105

KGX is a utility library and set of command line tools for exchanging data in Knowledge Graphs (KGs).

The tooling here is partly generic but intended primarily for building the translator-knowledge-graph, and thus expects KGs to be Biolink Model compliant.

The tool allows you to fetch (sub)graphs from one (or more) KG and create an entirely new KG.

The core data model is a Property Graph (PG), with the default representation using a networkx MultiDiGraph.

KGX supports Neo4j and RDF triple stores, along with other serialization formats such as TSV, CSV, JSON, JSON Lines, OBOGraph JSON, SSSOM, RDF NT, and OWL.

CHAPTER

ONE

CONTENTS

1.1 Installation

The installation for KGX requires Python 3.7 or greater.

1.1.1 Installation for users

Installing from PyPI

KGX is available on PyPI and can be installed using pip as follows,

```
pip install kgx
```

To install a particular version of KGX, be sure to specify the version number,

```
pip install kgx==0.5.0
```

Installing from GitHub

Clone the GitHub repository and then install,

```
git clone https://github.com/biolink/kgx
cd kgx
python setup.py install
```

1.1.2 Installation for developers

Setting up a development environment

To build directly from source, first clone the GitHub repository,

```
git clone https://github.com/biolink/kgx
cd kgx
```

Then install the necessary dependencies listed in `requirements.txt`,

```
pip3 install -r requirements.txt
```

For convenience, make use of the `venv` module in Python3 to create a lightweight virtual environment,

```
python3 -m venv env
source env/bin/activate

pip install -r requirements.txt
```

To install KGX you can do one of the following,

```
pip install .

# OR

python setup.py install
```

Setting up a testing environment

KGX has a suite of tests that rely on Docker containers to run Neo4j specific tests.

To set up the required containers, first install Docker on your local machine.

Once Docker is up and running, run the following commands:

```
docker run -d --name kgx-neo4j-integration-test \
           -p 7474:7474 -p 7687:7687 \
           --env NEO4J_AUTH=neo4j/test \
           neo4j:3.5.25
```

```
docker run -d --name kgx-neo4j-unit-test \
           -p 8484:7474 -p 8888:7687 \
           --env NEO4J_AUTH=neo4j/test \
           neo4j:3.5.25
```

Note: Setting up the Neo4j container is optional. If there is no container set up then the tests that rely on them are skipped.

KGX tests are simply run using `make`:

```
make tests
```

1.2 Reference

This section provides a detailed documentation on the classes and methods from all the submodules in KGX.

1.2.1 KGX Command Line Interface

The KGX CLI is a way of accessing KGX's functionality directly from the command line.

KGX CLI

kgx

Knowledge Graph Exchange CLI entrypoint.

```
kgx [OPTIONS] COMMAND [ARGS]...
```

Options

--version
Show the version and exit.

graph-summary

Loads and summarizes a knowledge graph from a set of input files.

```
kgx graph-summary [OPTIONS] INPUTS...
```

Options

-i, --input-format <input_format>
The input format. Can be one of ('tsv', 'csv', 'graph', 'json', 'jsonl', 'obojson', 'obo-json', 'trapi-json', 'neo4j', 'nt', 'owl', 'sssom') [required]

-c, --input-compression <input_compression>
The input compression type

-o, --output <output>
[required]

-r, --report-type <report_type>
The summary report type. Must be one of ('kgx-map', 'meta-knowledge-graph')

-f, --report-format <report_format>
The input format. Can be one of ('yaml', 'json')

-s, --stream
Parse input as a stream

-n, --graph-name <graph_name>
User specified name of graph being summarized (default: 'Graph')

--node-facet-properties <node_facet_properties>
A list of node properties from which to generate counts per value for those properties

--edge-facet-properties <edge_facet_properties>
A list of edge properties from which to generate counts per value for those properties

-l, --error-log <error_log>
File within which to report graph data parsing errors (default: "stderr")

Arguments

INPUTS
Required argument(s)

merge

Load nodes and edges from files and KGs, as defined in a config YAML, and merge them into a single graph. The merged graph can then be written to a local/remote Neo4j instance OR be serialized into a file.

```
kgx merge [OPTIONS]
```

Options

--merge-config <merge_config>
[required]

--source <source>
Source(s) from the YAML to process

--destination <destination>
Destination(s) from the YAML to process

-p, --processes <processes>
Number of processes to use

neo4j-download

Download nodes and edges from Neo4j database.

```
kgx neo4j-download [OPTIONS]
```

Options

-l, --uri <uri>
Neo4j URI to download from. For example, https://localhost:7474 [required]

-u, --username <username>
Neo4j username [required]

-p, --password <password>
Neo4j password [required]

-o, --output <output>
Output [required]

-f, --output-format <output_format>
The output format. Can be one of ('tsv', 'csv', 'graph', 'json', 'jsonl', 'obojson', 'obo-json', 'trapi-json', 'neo4j', 'nt', 'owl', 'sssom') [required]

-d, --output-compression <output_compression>
The output compression type

-s, --stream
Parse input as a stream

-n, --node-filters <node_filters>
Filters for filtering nodes from the input graph

-e, --edge-filters <edge_filters>
Filters for filtering edges from the input graph

neo4j-upload

Upload a set of nodes/edges to a Neo4j database.

```
kgx neo4j-upload [OPTIONS] INPUTS...
```

Options

-i, --input-format <input_format>
The input format. Can be one of ('tsv', 'csv', 'graph', 'json', 'jsonl', 'obojson', 'obo-json', 'trapi-json', 'neo4j', 'nt', 'owl', 'sssom') [required]

-c, --input-compression <input_compression>
The input compression type

-l, --uri <uri>
Neo4j URI to upload to. For example, https://localhost:7474 [required]

-u, --username <username>
Neo4j username [required]

-p, --password <password>
Neo4j password [required]

-s, --stream
Parse input as a stream

-n, --node-filters <node_filters>
Filters for filtering nodes from the input graph

-e, --edge-filters <edge_filters>
Filters for filtering edges from the input graph

Arguments

INPUTS
Required argument(s)

transform

Transform a Knowledge Graph from one serialization form to another.

```
kgx transform [OPTIONS] [INPUTS]...
```

Options

-i, --input-format <input_format>
The input format. Can be one of ('tsv', 'csv', 'graph', 'json', 'jsonl', 'obojson', 'obo-json', 'trapi-json', 'neo4j', 'nt', 'owl', 'sssom')

-c, --input-compression <input_compression>
The input compression type

-o, --output <output>
Output

-f, --output-format <output_format>
The output format. Can be one of ('tsv', 'csv', 'graph', 'json', 'jsonl', 'obojson', 'obo-json', 'trapi-json', 'neo4j', 'nt', 'owl', 'sssom')

-d, --output-compression <output_compression>
The output compression type

--stream
Parse input as a stream

-n, --node-filters <node_filters>
Filters for filtering nodes from the input graph

-e, --edge-filters <edge_filters>
Filters for filtering edges from the input graph

--transform-config <transform_config>
Transform config YAML

--source <source>
Source(s) from the YAML to process

-k, --knowledge-sources <knowledge_sources>
A named knowledge source with (string, boolean or tuple rewrite) specification

--infores-catalog <infores_catalog>
Optional dump of a CSV file of InfoRes CURIE to Knowledge Source mappings

-p, --processes <processes>
Number of processes to use

Arguments

INPUTS
Optional argument(s)

validate

Run KGX validator on an input file to check for Biolink Model compliance.

```
kgx validate [OPTIONS] INPUTS...
```

Options

-i, --input-format <input_format>
The input format. Can be one of ('tsv', 'csv', 'graph', 'json', 'jsonl', 'obojson', 'obo-json', 'trapi-json', 'neo4j', 'nt', 'owl', 'sssom') [required]

-c, --input-compression <input_compression>
The input compression type

-o, --output <output>
File to write validation reports to

-s, --stream
Parse input as a stream

-b, --biolink-release <biolink_release>
Biolink Model Release (SemVer) used for validation (default: latest Biolink Model Toolkit version)

Arguments

INPUTS
Required argument(s)

CLI Utils

Utility methods that are used in KGX command line.

kgx.cli.cli_utils

kgx.cli.cli_utils.**apply_operations**(*source: dict*, *graph: kgx.graph.base_graph.BaseGraph*) → kgx.graph.base_graph.BaseGraph
Apply operations as defined in the YAML.

Parameters

- **source** (*dict*) – The source from the YAML
- **graph** (kgx.graph.base_graph.BaseGraph) – The graph corresponding to the source

Returns The graph corresponding to the source

Return type *kgx.graph.base_graph.BaseGraph*

kgx.cli.cli_utils.**get_input_file_types**() → Tuple
Get all input file formats supported by KGX.

Returns A tuple of supported file formats

Return type Tuple

kgx.cli.cli_utils.**get_output_file_types**() → Tuple
Get all output file formats supported by KGX.

Returns A tuple of supported file formats

Return type Tuple

kgx.cli.cli_utils.**get_report_format_types**() → Tuple
Get all graph summary report formats supported by KGX.

Returns A tuple of supported file formats

Return type Tuple

kgx.cli.cli_utils.**graph_summary**(*inputs: List[str], input_format: str, input_compression: Optional[str], output: Optional[str], report_type: str, report_format: Optional[str] = None, stream: bool = False, graph_name: Optional[str] = None, node_facet_properties: Optional[List] = None, edge_facet_properties: Optional[List] = None, error_log: str = ''*) → Dict

Loads and summarizes a knowledge graph from a set of input files.

Parameters

- **inputs** (*List[str]*) – Input file
- **input_format** (*str*) – Input file format
- **input_compression** (*Optional[str]*) – The input compression type
- **output** (*Optional[str]*) – Where to write the output (stdout, by default)
- **report_type** (*str*) – The summary report type
- **report_format** (*Optional[str]*) – The summary report format file types: 'yaml' or 'json'
- **stream** (*bool*) – Whether to parse input as a stream
- **graph_name** (*str*) – User specified name of graph being summarized
- **node_facet_properties** (*Optional[List]*) – A list of node properties from which to generate counts per value for those properties. For example, `['provided_by']`
- **edge_facet_properties** (*Optional[List]*) – A list of edge properties (e.g. knowledge_source tags) to facet on. For example, `['original_knowledge_source', 'aggregator_knowledge_source']`
- **error_log** (*str*) – Where to write any graph processing error message (stderr, by default)

Returns A dictionary with the graph stats

Return type Dict

kgx.cli.cli_utils.**merge**(*merge_config: str, source: Optional[List] = None, destination: Optional[List] = None, processes: int = 1*) → kgx.graph.base_graph.BaseGraph

Load nodes and edges from files and KGs, as defined in a config YAML, and merge them into a single graph. The merged graph can then be written to a local/remote Neo4j instance OR be serialized into a file.

Parameters

- **merge_config** (*str*) – Merge config YAML
- **source** (*Optional[List]*) – A list of source to load from the YAML
- **destination** (*Optional[List]*) – A list of destination to write to, as defined in the YAML
- **processes** (*int*) – Number of processes to use

Returns The merged graph

Return type *kgx.graph.base_graph.BaseGraph*

kgx.cli.cli_utils.**neo4j_download**(*uri: str, username: str, password: str, output: str, output_format: str, output_compression: Optional[str], stream: bool, node_filters: Optional[Tuple] = None, edge_filters: Optional[Tuple] = None*) → kgx.transformer.Transformer

Download nodes and edges from Neo4j database.

Parameters

- **uri** (*str*) – Neo4j URI. For example, https://localhost:7474
- **username** (*str*) – Username for authentication
- **password** (*str*) – Password for authentication
- **output** (*str*) – Where to write the output (stdout, by default)
- **output_format** (*Optional[str]*) – The output type (`tsv`, by default)
- **output_compression** (*Optional[str]*) – The output compression type
- **stream** (*bool*) – Whether to parse input as a stream
- **node_filters** (*Optional[Tuple]*) – Node filters
- **edge_filters** (*Optional[Tuple]*) – Edge filters

Returns The NeoTransformer

Return type kgx.Transformer

kgx.cli.cli_utils.**neo4j_upload**(*inputs: List[str], input_format: str, input_compression: Optional[str], uri: str, username: str, password: str, stream: bool, node_filters: Optional[Tuple] = None, edge_filters: Optional[Tuple] = None*) → kgx.transformer.Transformer

Upload a set of nodes/edges to a Neo4j database.

Parameters

- **inputs** (*List[str]*) – A list of files that contains nodes/edges
- **input_format** (*str*) – The input format
- **input_compression** (*Optional[str]*) – The input compression type
- **uri** (*str*) – The full HTTP address for Neo4j database
- **username** (*str*) – Username for authentication
- **password** (*str*) – Password for authentication
- **stream** (*bool*) – Whether to parse input as a stream
- **node_filters** (*Optional[Tuple]*) – Node filters
- **edge_filters** (*Optional[Tuple]*) – Edge filters

Returns The NeoTransformer

Return type kgx.Transformer

kgx.cli.cli_utils.**parse_source**(*key: str, source: dict, output_directory: str, prefix_map: Dict[str, str] = None, node_property_predicates: Set[str] = None, predicate_mappings: Dict[str, str] = None, checkpoint: bool = False*) → kgx.sink.sink.Sink

Parse a source from a merge config YAML.

Parameters

- **key** (*str*) – Source key
- **source** (*Dict*) – Source configuration
- **output_directory** (*str*) – Location to write output to
- **prefix_map** (*Dict[str, str]*) – Non-canonical CURIE mappings
- **node_property_predicates** (*Set[str]*) – A set of predicates that ought to be treated as node properties (This is applicable for RDF)
- **predicate_mappings** (*Dict[str, str]*) – A mapping of predicate IRIs to property names (This is applicable for RDF)
- **checkpoint** (*bool*) – Whether to serialize each individual source to a TSV

Returns Returns an instance of Sink

Return type *kgx.sink.sink.Sink*

kgx.cli.cli_utils.**prepare_input_args**(*key: str, source: Dict, output_directory: Optional[str], prefix_map: Dict[str, str] = None, node_property_predicates: Set[str] = None, predicate_mappings: Dict[str, str] = None*) → Dict

Prepare input arguments for Transformer.

Parameters

- **key** (*str*) – Source key
- **source** (*Dict*) – Source configuration
- **output_directory** (*str*) – Location to write output to
- **prefix_map** (*Dict[str, str]*) – Non-canonical CURIE mappings
- **node_property_predicates** (*Set[str]*) – A set of predicates that ought to be treated as node properties (This is applicable for RDF)
- **predicate_mappings** (*Dict[str, str]*) – A mapping of predicate IRIs to property names (This is applicable for RDF)

Returns Input arguments as dictionary

Return type Dict

kgx.cli.cli_utils.**prepare_output_args**(*key: str, source: Dict, output_directory: Optional[str], reverse_prefix_map: Dict = None, reverse_predicate_mappings: Dict = None, property_types: Dict = None*) → Dict

Prepare output arguments for Transformer.

Parameters

- **key** (*str*) – Source key
- **source** (*Dict*) – Source configuration
- **output_directory** (*str*) – Location to write output to
- **reverse_prefix_map** (*Dict[str, str]*) – Non-canonical CURIE mappings for export
- **reverse_predicate_mappings** (*Dict[str, str]*) – A mapping of property names to predicate IRIs (This is applicable for RDF)

- **property_types** (*Dict[str, str]*) – The xml property type for properties that are other than `xsd:string`. Relevant for RDF export.

Returns Output arguments as dictionary

Return type Dict

kgx.cli.cli_utils.**prepare_top_level_args**(*d: Dict*) → Dict
Parse top-level configuration.

Parameters **d** (*Dict*) – The configuration section from the transform/merge YAML

Returns A parsed dictionary with parameters from configuration

Return type Dict

kgx.cli.cli_utils.**transform**(*inputs: Optional[List[str]], input_format: Optional[str] = None, input_compression: Optional[str] = None, output: Optional[str] = None, output_format: Optional[str] = None, output_compression: Optional[str] = None, stream: bool = False, node_filters: Optional[List[Tuple[str, str]]] = None, edge_filters: Optional[List[Tuple[str, str]]] = None, transform_config: str = None, source: Optional[List] = None, knowledge_sources: Optional[List[Tuple[str, str]]] = None, processes: int = 1, infores_catalog: Optional[str] = None*) → None
Transform a Knowledge Graph from one serialization form to another.

Parameters

- **inputs** (*Optional[List[str]]*) – A list of files that contains nodes/edges
- **input_format** (*Optional[str]*) – The input format
- **input_compression** (*Optional[str]*) – The input compression type
- **output** (*Optional[str]*) – The output file
- **output_format** (*Optional[str]*) – The output format
- **output_compression** (*Optional[str]*) – The output compression type
- **stream** (*bool*) – Whether to parse input as a stream
- **node_filters** (*Optional[List[Tuple[str, str]]]*) – Node input filters
- **edge_filters** (*Optional[List[Tuple[str, str]]]*) – Edge input filters
- **transform_config** (*Optional[str]*) – The transform config YAML
- **source** (*Optional[List]*) – A list of source to load from the YAML
- **knowledge_sources** (*Optional[List[Tuple[str, str]]]*) – A list of named knowledge sources with (string, boolean or tuple rewrite) specification
- **processes** (*int*) – Number of processes to use
- **infores_catalog** (*Optional[str]*) – Optional dump of a TSV file of InfoRes CURIE to Knowledge Source mappings (not yet available in transform_config calling mode)

kgx.cli.cli_utils.**transform_source**(*key: str, source: Dict, output_directory: Optional[str], prefix_map: Dict[str, str] = None, node_property_predicates: Set[str] = None, predicate_mappings: Dict[str, str] = None, reverse_prefix_map: Dict = None, reverse_predicate_mappings: Dict = None, property_types: Dict = None, checkpoint: bool = False, preserve_graph: bool = True, stream: bool = False, infores_catalog: Optional[str] = None*) → kgx.sink.sink.Sink

Transform a source from a transform config YAML.

Parameters

- **key** (*str*) – Source key
- **source** (*Dict*) – Source configuration
- **output_directory** (*Optional[str]*) – Location to write output to
- **prefix_map** (*Dict[str, str]*) – Non-canonical CURIE mappings
- **node_property_predicates** (*Set[str]*) – A set of predicates that ought to be treated as node properties (This is applicable for RDF)
- **predicate_mappings** (*Dict[str, str]*) – A mapping of predicate IRIs to property names (This is applicable for RDF)
- **reverse_prefix_map** (*Dict[str, str]*) – Non-canonical CURIE mappings for export
- **reverse_predicate_mappings** (*Dict[str, str]*) – A mapping of property names to predicate IRIs (This is applicable for RDF)
- **property_types** (*Dict[str, str]*) – The xml property type for properties that are other than `xsd:string`. Relevant for RDF export.
- **checkpoint** (*bool*) – Whether to serialize each individual source to a TSV
- **preserve_graph** (*true*) – Whether or not to preserve the graph corresponding to the source
- **stream** (*bool*) – Whether to parse input as a stream
- **infores_catalog** (*Optional[str]*) – Optional dump of a TSV file of InfoRes CURIE to Knowledge Source mappings

Returns Returns an instance of Sink

Return type *kgx.sink.sink.Sink*

kgx.cli.cli_utils.**validate**(*inputs: List[str], input_format: str, input_compression: Optional[str], output: Optional[str], stream: bool, biolink_release: Optional[str] = None*) → List

Run KGX validator on an input file to check for Biolink Model compliance.

Parameters

- **inputs** (*List[str]*) – Input files
- **input_format** (*str*) – The input format
- **input_compression** (*Optional[str]*) – The input compression type
- **output** (*Optional[str]*) – Path to output file (stdout, by default)
- **stream** (*bool*) – Whether to parse input as a stream.

- **biolink_release** (*Optional[str] = None*) – SemVer version of Biolink Model Release used for validation (default: latest Biolink Model Toolkit version)

Returns Returns a list of errors, if any

Return type List

1.2.2 Graph

KGX makes use of an in-memory labelled property graph for representing a Knowledge Graph.

To support a wide variety of graph libraries, KGX has a Graph API which abstracts over the underlying graph store.

Should you want to add support for a new graph store,

- create a new class that extends `kgx.graph.base_graph.BaseGraph`.
- modify the `graph_store` variable in kgx/config.yml.

kgx.graph.base_graph.BaseGraph

`BaseGraph` is the base Graph API that can be used to abstract over any graph, as long as the graph is capable of successfully representing a property graph.

class kgx.graph.base_graph.**BaseGraph**

Bases: `object`

BaseGraph that is a wrapper and provides methods to interact with a graph store.

All implementations should extend this BaseGraph class and implement all the defined methods.

add_edge(*subject_node: str*, *object_node: str*, *edge_key: Optional[str] = None*, ***kwargs: Any*) → Any

Add an edge to the graph.

Parameters

- **subject_node** (*str*) – The subject (source) node
- **object_node** (*str*) – The object (target) node
- **edge_key** (*Optional[str]*) – The edge key
- **kwargs** (*Any*) – Any additional edge properties

Returns

Return type Any

add_edge_attribute(*subject_node: str, object_node: str, edge_key: Optional[str], attr_key: str, attr_value: Any*) → Any

Add an attribute to a given edge.

Parameters

- **subject_node** (*str*) – The subject (source) node
- **object_node** (*str*) – The object (target) node
- **edge_key** (*Optional[str]*) – The edge key
- **attr_key** (*str*) – The attribute key
- **attr_value** (*Any*) – The attribute value

Returns

Return type Any

add_node(*node: str*, ***kwargs: Any*) → Any
Add a node to the graph.

Parameters

- **node** (*str*) – Node identifier
- ****kwargs** (*Any*) – Any additional node properties

add_node_attribute(*node: str*, *key: str*, *value: Any*) → Any
Add an attribute to a given node.

Parameters

- **node** (*str*) – The node identifier
- **key** (*str*) – The key for an attribute
- **value** (*Any*) – The value corresponding to the key

Returns

Return type Any

clear() → None
Remove all the nodes and edges in the graph.

degree()
Get the degree of all the nodes in a graph.

edges(*keys: bool = False*, *data: bool = True*) → Dict
Get all edges in a graph.

Parameters

- **keys** (*bool*) – Whether or not to include edge keys
- **data** (*bool*) – Whether or not to fetch node properties

Returns A dictionary of edges

Return type Dict

edges_iter() → Generator
Get an iterable to traverse through all the edges in a graph.

Returns A generator for edges

Return type Generator

get_edge(*subject_node: str, object_node: str, edge_key: Optional[str]*) → Dict
Get an edge and its properties.

Parameters

- **subject_node** (*str*) – The subject (source) node
- **object_node** (*str*) – The object (target) node
- **edge_key** (*Optional[str]*) – The edge key

Returns The edge dictionary

Return type Dict

static get_edge_attributes(*graph: Any*, *attr_key: str*) → Any
Get all edges that have a value for the given attribute `attr_key`.

Parameters

- **graph** (*Any*) – The graph to modify
- **attr_key** (*str*) – The attribute key

Returns

Return type Any

get_node(*node: str*) → Dict
Get a node and its properties.

Parameters **node** (*str*) – The node identifier

Returns The node dictionary

Return type Dict

static get_node_attributes(*graph: Any*, *attr_key: str*) → Any
Get all nodes that have a value for the given attribute `attr_key`.

Parameters

- **graph** (*Any*) – The graph to modify
- **attr_key** (*str*) – The attribute key

Returns

Return type Any

has_edge(*subject_node: str*, *object_node: str*, *edge_key: Optional[str] = None*) → bool
Check whether a given edge exists in the graph.

Parameters

- **subject_node** (*str*) – The subject (source) node
- **object_node** (*str*) – The object (target) node
- **edge_key** (*Optional[str]*) – The edge key

Returns Whether or not the given edge exists

Return type bool

has_node(*node: str*) → bool
Check whether a given node exists in the graph.

Parameters **node** (*str*) – The node identifier

Returns Whether or not the given node exists

Return type bool

in_edges(*node: str*, *keys: bool = False*, *data: bool = False*) → List
Get all incoming edges for a given node.

Parameters

- **node** (*str*) – The node identifier
- **keys** (*bool*) – Whether or not to include edge keys
- **data** (*bool*) – Whether or not to fetch node properties

Returns A list of edges

Return type List

nodes(*data: bool = True*) → Dict
Get all nodes in a graph.

Parameters **data** (*bool*) – Whether or not to fetch node properties

Returns A dictionary of nodes

Return type Dict

nodes_iter() → Generator
Get an iterable to traverse through all the nodes in a graph.

Returns A generator for nodes

Return type Generator

number_of_edges() → int
Returns the number of edges in a graph.

Returns

Return type int

number_of_nodes() → int
Returns the number of nodes in a graph.

Returns

Return type int

out_edges(*node: str*, *keys: bool = False*, *data: bool = False*) → List
Get all outgoing edges for a given node.

Parameters

- **node** (*str*) – The node identifier
- **keys** (*bool*) – Whether or not to include edge keys
- **data** (*bool*) – Whether or not to fetch node properties

Returns A list of edges

Return type List

static relabel_nodes(*graph: Any*, *mapping: Dict*) → Any
Relabel identifiers for a series of nodes based on mappings.

Parameters

- **graph** (*Any*) – The graph to modify
- **mapping** (*Dict[str, str]*) – A dictionary of mapping where the key is the old identifier and the value is the new identifier.

Returns

Return type Any

remove_edge(*subject_node: str*, *object_node: str*, *edge_key: Optional[str] = None*) → Any
Remove a given edge from the graph.

Parameters

- **subject_node** (*str*) – The subject (source) node

- **object_node** (*str*) – The object (target) node
- **edge_key** (*Optional[str]*) – The edge key

Returns

Return type Any

remove_node(*node: str*) → Any
Remove a given node from the graph.

Parameters **node** (*str*) – The node identifier

Returns

Return type Any

static set_edge_attributes(*graph: Any*, *attributes: Dict*) → Any
Set nodes attributes from a dictionary of key-values.

Parameters

- **graph** (*Any*) – The graph to modify
- **attributes** (*Dict*) – A dictionary of node identifier to key-value pairs

Returns

Return type Any

static set_node_attributes(*graph: Any*, *attributes: Dict*) → Any
Set nodes attributes from a dictionary of key-values.

Parameters

- **graph** (*Any*) – The graph to modify
- **attributes** (*Dict*) – A dictionary of node identifier to key-value pairs

Returns

Return type Any

update_edge_attribute(*subject_node: str, object_node: str, edge_key: Optional[str], attr_key: str, attr_value: Any*) → Dict
Update an attribute of a given edge.

Parameters

- **subject_node** (*str*) – The subject (source) node
- **object_node** (*str*) – The object (target) node
- **edge_key** (*Optional[str]*) – The edge key
- **attr_key** (*str*) – The attribute key
- **attr_value** (*Any*) – The attribute value

Returns A dictionary corresponding to the updated edge properties

Return type Dict

update_node_attribute(*node*, *key: str*, *value: Any*) → Dict
Update an attribute of a given node.

Parameters

- **node** (*str*) – The node identifier

- **key** (*str*) – The key for an attribute
- **value** (*Any*) – The value corresponding to the key

Returns A dictionary corresponding to the updated node properties

Return type Dict

kgx.graph.nx_graph.NxGraph

NxGraph is basically an abstraction on top of networkx.MultiDiGraph.

The NxGraph subclasses `kgx.graph.base_graph.BaseGraph` and implements all the methods defined in `BaseGraph`.

class kgx.graph.nx_graph.**NxGraph**
Bases: *kgx.graph.base_graph.BaseGraph*

NxGraph is a wrapper that provides methods to interact with a networkx.MultiDiGraph.

NxGraph extends kgx.graph.base_graph.BaseGraph and implements all the methods from BaseGraph.

add_edge(*subject_node: str*, *object_node: str*, *edge_key: str = None*, ***kwargs: Any*) → None
Add an edge to the graph.

Parameters

- **subject_node** (*str*) – The subject (source) node
- **object_node** (*str*) – The object (target) node
- **edge_key** (*Optional[str]*) – The edge key
- **kwargs** (*Any*) – Any additional edge properties

add_edge_attribute(*subject_node: str, object_node: str, edge_key: Optional[str], attr_key: str, attr_value: Any*) → None
Add an attribute to a given edge.

Parameters

- **subject_node** (*str*) – The subject (source) node
- **object_node** (*str*) – The object (target) node
- **edge_key** (*Optional[str]*) – The edge key
- **attr_key** (*str*) – The attribute key
- **attr_value** (*Any*) – The attribute value

add_node(*node: str*, ***kwargs: Any*) → None
Add a node to the graph.

Parameters

- **node** (*str*) – Node identifier
- ****kwargs** (*Any*) – Any additional node properties

add_node_attribute(*node: str*, *attr_key: str*, *attr_value: Any*) → None
Add an attribute to a given node.

Parameters

- **node** (*str*) – The node identifier

- **attr_key** (*str*) – The key for an attribute
- **attr_value** (*Any*) – The value corresponding to the key

clear() → None
Remove all the nodes and edges in the graph.

degree()
Get the degree of all the nodes in a graph.

edges(*keys: bool = False*, *data: bool = True*) → Dict
Get all edges in a graph.

Parameters

- **keys** (*bool*) – Whether or not to include edge keys
- **data** (*bool*) – Whether or not to fetch node properties

Returns A dictionary of edges

Return type Dict

edges_iter() → Generator
Get an iterable to traverse through all the edges in a graph.

Returns A generator for edges where each element is a 4-tuple that contains (subject, object, edge_key, edge_data)

Return type Generator

get_edge(*subject_node: str*, *object_node: str*, *edge_key: Optional[str] = None*) → Dict
Get an edge and its properties.

Parameters

- **subject_node** (*str*) – The subject (source) node
- **object_node** (*str*) – The object (target) node
- **edge_key** (*Optional[str]*) – The edge key

Returns The edge dictionary

Return type Dict

static get_edge_attributes(*graph: kgx.graph.base_graph.BaseGraph*, *attr_key: str*) → Dict
Get all edges that have a value for the given attribute `attr_key`.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph to modify
- **attr_key** (*str*) – The attribute key

Returns A dictionary where edges are the keys and the values are the attribute values for `attr_key`

Return type Dict

get_node(*node: str*) → Dict
Get a node and its properties.

Parameters **node** (*str*) – The node identifier

Returns The node dictionary

Return type Dict

static get_node_attributes (*graph: kgx.graph.base_graph.BaseGraph*, *attr_key: str*) → Dict
Get all nodes that have a value for the given attribute `attr_key`.

Parameters

- **graph** (`kgx.graph.base_graph.BaseGraph`) – The graph to modify
- **attr_key** (*str*) – The attribute key

Returns A dictionary where nodes are the keys and the values are the attribute values for `key`

Return type Dict

has_edge (*subject_node: str*, *object_node: str*, *edge_key: Optional[str] = None*) → bool
Check whether a given edge exists in the graph.

Parameters

- **subject_node** (*str*) – The subject (source) node
- **object_node** (*str*) – The object (target) node
- **edge_key** (*Optional[str]*) – The edge key

Returns Whether or not the given edge exists

Return type bool

has_node (*node: str*) → bool
Check whether a given node exists in the graph.

Parameters **node** (*str*) – The node identifier

Returns Whether or not the given node exists

Return type bool

in_edges (*node: str*, *keys: bool = False*, *data: bool = False*) → List
Get all incoming edges for a given node.

Parameters

- **node** (*str*) – The node identifier
- **keys** (*bool*) – Whether or not to include edge keys
- **data** (*bool*) – Whether or not to fetch node properties

Returns A list of edges

Return type List

nodes (*data: bool = True*) → Dict
Get all nodes in a graph.

Parameters **data** (*bool*) – Whether or not to fetch node properties

Returns A dictionary of nodes

Return type Dict

nodes_iter () → Generator
Get an iterable to traverse through all the nodes in a graph.

Returns A generator for nodes where each element is a Tuple that contains (node_id, node_data)

Return type Generator

number_of_edges() → int
Returns the number of edges in a graph.

Returns

Return type int

number_of_nodes() → int
Returns the number of nodes in a graph.

Returns

Return type int

out_edges(*node: str*, *keys: bool = False*, *data: bool = False*) → List
Get all outgoing edges for a given node.

Parameters

- **node** (*str*) – The node identifier
- **keys** (*bool*) – Whether or not to include edge keys
- **data** (*bool*) – Whether or not to fetch node properties

Returns A list of edges

Return type List

static relabel_nodes(*graph: kgx.graph.base_graph.BaseGraph*, *mapping: Dict*) → None
Relabel identifiers for a series of nodes based on mappings.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph to modify
- **mapping** (*Dict*) – A dictionary of mapping where the key is the old identifier and the value is the new identifier.

remove_edge(*subject_node: str*, *object_node: str*, *edge_key: Optional[str] = None*) → None
Remove a given edge from the graph.

Parameters

- **subject_node** (*str*) – The subject (source) node
- **object_node** (*str*) – The object (target) node
- **edge_key** (*Optional[str]*) – The edge key

remove_node(*node: str*) → None
Remove a given node from the graph.

Parameters **node** (*str*) – The node identifier

static set_edge_attributes(*graph: kgx.graph.base_graph.BaseGraph*, *attributes: Dict*) → None
Set nodes attributes from a dictionary of key-values.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph to modify
- **attributes** (*Dict*) – A dictionary of node identifier to key-value pairs

Returns

Return type Any

static set_node_attributes(*graph: kgx.graph.base_graph.BaseGraph*, *attributes: Dict*) → None

Set nodes attributes from a dictionary of key-values.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph to modify
- **attributes** (*Dict*) – A dictionary of node identifier to key-value pairs

update_edge_attribute(*subject_node: str, object_node: str, edge_key: Optional[str], attr_key: str, attr_value: Any, preserve: bool = False*) → Dict

Update an attribute of a given edge.

Parameters

- **subject_node** (*str*) – The subject (source) node
- **object_node** (*str*) – The object (target) node
- **edge_key** (*Optional[str]*) – The edge key
- **attr_key** (*str*) – The attribute key
- **attr_value** (*Any*) – The attribute value
- **preserve** (*bool*) – Whether or not to preserve existing values for the given attr_key

Returns A dictionary corresponding to the updated edge properties

Return type Dict

update_node_attribute(*node: str*, *attr_key: str*, *attr_value: Any*, *preserve: bool = False*) → Dict

Update an attribute of a given node.

Parameters

- **node** (*str*) – The node identifier
- **attr_key** (*str*) – The key for an attribute
- **attr_value** (*Any*) – The value corresponding to the key
- **preserve** (*bool*) – Whether or not to preserve existing values for the given attr_key

Returns A dictionary corresponding to the updated node properties

Return type Dict

1.2.3 Transformer

The Transformer class is responsible for connecting a source to a sink where records are read from the source and written to a sink.

The Transformer supports two modes:

- No streaming
- Streaming

No streaming

In this mode, the Transformer reads records from a source and writes to an intermediate graph. One can then use this intermediate graph as a substrate for various graph operations.

```
from kgx.transformer import Transformer

input_args = {'filename': ['graph_nodes.tsv', 'graph_edges.tsv'], 'format': 'tsv'}
output_args = {'filename': 'graph.json', 'format': 'json'}

t = Transformer(stream=False)

# read from TSV
t.transform(input_args=input_args)

# The intermediate graph store can be accessed via t.store.graph

# write to JSON
t.save(output_args=output_args)
```

Streaming

In this mode, records are read from a source and written to sink, on-the-fly.

```
from kgx.transformer import Transformer

input_args = {'filename': ['graph_nodes.tsv', 'graph_edges.tsv'], 'format': 'tsv'}
output_args = {'filename': 'graph.json', 'format': 'json'}

t = Transformer(stream=True)

# read from TSV and write to JSON
t.transform(input_args=input_args, output_args=output_args)
```

Inspecting the Knowledge Data Flow

Note that `transform` operation accepts an optional inspect *Callable* argument which injects node/edge data stream inspection into the `Transform.process` operation of `Transform.transform` operations. See the unit test module in the KGX project tests/integration/test_transform.py for an example of usage of this callable argument.

This feature, when coupled with the `--stream` and a 'null' Transformer Sink (i.e. `output_args = {'format': 'null'}`), allows "just-in-time" processing of the nodes and edges of huge graphs without incurring a large in-memory footprint.

Provenance of Nodes and Edges

Biolink Model 2.0 specified new properties for edge provenance to replace the (now deprecated) `provided_by` provenance property (the `provided_by` property may still be used for node annotation).

One or more of these provenance properties may optionally be inserted as dictionary entries into the input arguments to specify default global values for these properties. Such values will be used when an edge lacks an explicit provenance property. If one does not specify such a global property, then the algorithm heuristically infers and sets a default `knowledge_source` value.

```
from kgx.transformer import Transformer

input_args = {
    filename': [
            'graph_nodes.tsv',
            'graph_edges.tsv'],
```

(continues on next page)

(continued from previous page)

```
    'format': 'tsv',
    'provided_by': "My Test Source",
    'aggregator_knowledge_source': "My Test Source"

}

t = Transformer()

# read from TSV
t.transform(input_args=input_args)

# use the transformed graph
t.store.graph.nodes()
t.store.graph.edges()
```

InfoRes Identifier Rewriting

The `provided_by` and/or `knowledge_source` *et al.* field values of KGX node and edge records generally contain a name of a knowledge source for the node or edge. In some cases, (e.g. Monarch) such values in source knowledge sources could be quite verbose. To normalize such names to a concise standard, the latest Biolink Model (2.*) is moving towards the use of **Information Resource** ("InfoRes") CURIE identifiers.

To help generate and document such InfoRes identifiers, the provenance property values may optionally trigger a rewrite of their knowledge source names to a candidate InfoRes, as follows:

1. Setting the provenance property to a boolean **True* or (case insensitive) string **"True"** triggers a simple reformatting of knowledge source names into lower case alphanumeric strings removing non-alphanumeric characters and replacing space delimiting words, with hyphens.
2. Setting the provenance property to a boolean **False* or (case insensitive) string **"False"** suppresses the given provenance annotation on the output graph.
3. Providing a tuple with a single string argument not equal to **True**, then the string assumed to be a standard (Pythonic) regular expression to match against knowledge source names. If you do not provide any other string argument (see below), then a matching substring in the name triggers deletion of the matched patter. The simple reformatting (as in 1 above) is then applied to the resulting string.
4. Similar to 2 above, except providing a second string in the tuple which is substituted for the regular expression matched string, followed by simple reformatting.
5. Providing a third string in the tuple to add a prefix string to the name (as a separate word) of all the generated InfoRes identifiers. Note that if one sets the first and second elements of the tuple to empty strings, the result is the simple addition of a prefix to the provenance property value. Again, the algorithm then applies the simple reformatting rules, but no other internal changes.

The unit tests provide examples of these various rewrites, in the KGX project tests/integration/test_transform.py.

The catalog of inferred InfoRes mappings onto knowledge source names is available programmatically, after completion of transform call by using the `get_infores_catalog()` method of the **Transformer** class. The `transform` call of the CLI now also takes a multi-valued `--knowledge-sources` argument, which either facilitates the aforementioned infores processing. Note that quoted comma-delimited strings demarcate the tuple rewrite specifications noted above.

kgx.transformer

class kgx.transformer.**Transformer**(*stream: bool = False*, *infores_catalog: Optional[str] = None*)

Bases: object

The Transformer class is responsible for transforming data from one form to another.

Parameters

- **stream** (*bool*) – Whether or not to stream
- **infores_catalog** (*Optional[str]*) – Optional dump of a TSV file of InfoRes CURIE to Knowledge Source mappings

get_infores_catalog()

Return catalog of Information Resource mappings aggregated from all Transformer associated sources

get_sink(***kwargs: Dict*) → kgx.sink.sink.Sink
Get an instance of Sink that corresponds to a given format.

Parameters **kwargs** (*Dict*) – Arguments required for initializing an instance of Sink

Returns An instance of kgx.sink.Sink

Return type *Sink*

get_source(*format: str*) → kgx.source.source.Source
Get an instance of Source that corresponds to a given format.

Parameters **format** (*str*) – The input store format

Returns An instance of kgx.source.Source

Return type *Source*

process(*source: Generator*, *sink: kgx.sink.sink.Sink*) → None
This method is responsible for reading from source and writing to sink by calling the relevant methods based on the incoming data.

Note: The streamed data must not be mutated.

Parameters

- **source** (*Generator*) – A generator from a Source
- **sink** (kgx.sink.sink.Sink) – An instance of Sink

save(*output_args: Dict*) → None
Save data from the in-memory store to a desired sink.

Parameters **output_args** (*Dict*) – Arguments relevant to your output sink

transform(*input_args: Dict, output_args: Optional[Dict] = None, inspector: Optional[Callable[[kgx.utils.kgx_utils.GraphEntityType, List], None]] = None*) → None
Transform an input source and write to an output sink.

If output_args is not defined then the data is persisted to an in-memory graph.

The 'inspector' argument is an optional Callable which the transformer.process() method applies to 'inspect' source records prior to writing them out to the Sink. The first (GraphEntityType) argument of the

Callable tags the record as a NODE or an EDGE. The second argument given to the Callable is the current record itself. This Callable is strictly meant to be procedural and should *not* mutate the record.

Parameters

- **input_args** (*Dict*) – Arguments relevant to your input source
- **output_args** (*Optional[Dict]*) – Arguments relevant to your output sink (
- **inspector** (*Optional[Callable[[GraphEntityType, List], None]]*) – Optional Callable to 'inspect' source records during processing.

1.2.4 Source

A Source can be implemented for any file, local, and/or remote store that can contains a graph. A Source is responsible for reading nodes and edges from the graph.

A source must subclass `kgx.source.source.Source` class and must implement the following methods:

- `parse`
- `read_nodes`
- `read_edges`

`parse` method

- Responsible for parsing a graph from a file/store
- Must return a generator that iterates over list of node and edge records from the graph

`read_nodes` method

- Responsible for reading nodes from the file/store
- Must return a generator that iterates over list of node records
- Each node record must be a 2-tuple `(node_id, node_data)` where,
  - `node_id` is the node CURIE
  - `node_data` is a dictionary that represents the node properties

`read_edges` method

- Responsible for reading edges from the file/store
- Must return a generator that iterates over list of edge records
- Each edge record must be a 4-tuple `(subject_id, object_id, edge_key, edge_data)` where,
  - `subject_id` is the subject node CURIE
  - `object_id` is the object node CURIE
  - `edge_key` is the unique key for the edge
  - `edge_data` is a dictionary that represents the edge properties

kgx.source.source

Base class for all Sources in KGX.

class kgx.source.source.**Source**
Bases: object

A Source is responsible for reading data as records from a store where the store is a file or a database.

check_edge_filter(*edge: Dict*) → bool
Check if an edge passes defined edge filters.

Parameters **edge** (*Dict*) – An edge

Returns Whether the given edge has passed all defined edge filters

Return type bool

check_node_filter(*node: Dict*) → bool
Check if a node passes defined node filters.

Parameters **node** (*Dict*) – A node

Returns Whether the given node has passed all defined node filters

Return type bool

clear_graph_metadata()
Clears a Source graph's internal graph_metadata. The value of such graph metadata is (now) generally a Callable function. This operation can be used in the code when the metadata is no longer needed, but may cause peculiar Python object persistent problems downstream.

get_infores_catalog() → Dict[str, str]
Return the InfoRes Context of the source

set_edge_filter(*key: str*, *value: set*) → None
Set an edge filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'subject_category' or 'object_category' filter, the value should be of type `set`. This method also sets the 'category' node filter, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for edge filter
- **value** (*Union[str, set]*) – The value for the edge filter. Can be either a string or a set.

set_edge_filters(*filters: Dict*) → None
Set edge filters.

Parameters **filters** (*Dict*) – Edge filters

set_edge_provenance(*edge_data*)
Set a specific edge provenance value.

set_node_filter(*key: str, value: Union[str, set]*) → None
Set a node filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'category' filter, the value should be of type `set`. This method also sets the 'subject_category' and 'object_category' edge filters, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for node filter
- **value** (*Union[str, set]*) – The value for the node filter. Can be either a string or a set.

set_node_filters (*filters: Dict*) → None
Set node filters.

Parameters **filters** (*Dict*) – Node filters

set_node_provenance (*node_data*)
Set a specific node provenance value.

set_prefix_map (*m: Dict*) → None
Update default prefix map.

Parameters **m** (*Dict*) – A dictionary with prefix to IRI mappings

set_provenance_map (*kwargs*)
Set up a provenance (Knowledge Source to InfoRes) map

kgx.source.graph_source

`GraphSource` is responsible for reading from an instance of `kgx.graph.base_graph.BaseGraph` and must use only the methods exposed by `BaseGraph` to access the graph.

class kgx.source.graph_source.**GraphSource**
Bases: *kgx.source.source.Source*

GraphSource is responsible for reading data as records from an in memory graph representation.

The underlying store must be an instance of `kgx.graph.base_graph.BaseGraph`

check_edge_filter (*edge: Dict*) → bool
Check if an edge passes defined edge filters.

Parameters **edge** (*Dict*) – An edge

Returns Whether the given edge has passed all defined edge filters

Return type bool

check_node_filter (*node: Dict*) → bool
Check if a node passes defined node filters.

Parameters **node** (*Dict*) – A node

Returns Whether the given node has passed all defined node filters

Return type bool

clear_graph_metadata ()
Clears a Source graph's internal graph_metadata. The value of such graph metadata is (now) generally a Callable function. This operation can be used in the code when the metadata is no longer needed, but may cause peculiar Python object persistent problems downstream.

get_infores_catalog() → Dict[str, str]
Return the InfoRes Context of the source

parse(*graph: kgx.graph.base_graph.BaseGraph*, ***kwargs: Any*) → Generator
This method reads from a graph and yields records.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph to read from
- **kwargs** (*Any*) – Any additional arguments

Returns A generator for node and edge records read from the graph

Return type Generator

read_edges() → Generator
Read edges as records from the graph.

Returns A generator for edges

Return type Generator

read_nodes() → Generator
Read nodes as records from the graph.

Returns A generator for nodes

Return type Generator

set_edge_filter(*key: str*, *value: set*) → None
Set an edge filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'subject_category' or 'object_category' filter, the value should be of type `set`. This method also sets the 'category' node filter, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for edge filter
- **value** (*Union[str, set]*) – The value for the edge filter. Can be either a string or a set.

set_edge_filters(*filters: Dict*) → None
Set edge filters.

Parameters **filters** (*Dict*) – Edge filters

set_edge_provenance(*edge_data*)
Set a specific edge provenance value.

set_node_filter(*key: str, value: Union[str, set]*) → None
Set a node filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'category' filter, the value should be of type `set`. This method also sets the 'subject_category' and 'object_category' edge filters, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for node filter
- **value** (*Union[str, set]*) – The value for the node filter. Can be either a string or a set.

set_node_filters(*filters: Dict*) → None
Set node filters.

Parameters **filters** (*Dict*) – Node filters

set_node_provenance(*node_data*)
Set a specific node provenance value.

set_prefix_map(*m: Dict*) → None
Update default prefix map.

Parameters **m** (*Dict*) – A dictionary with prefix to IRI mappings

set_provenance_map(*kwargs*)
Set up a provenance (Knowledge Source to InfoRes) map

kgx.source.tsv_source

`TsvSource` is responsible for reading from KGX formatted CSV or TSV using Pandas where every flat file is treated as a Pandas DataFrame and from which data are read in chunks.

KGX expects two separate files - one for nodes and another for edges.

class kgx.source.tsv_source.**TsvSource**
Bases: *kgx.source.source.Source*

TsvSource is responsible for reading data as records from a TSV/CSV.

check_edge_filter(*edge: Dict*) → bool
Check if an edge passes defined edge filters.

Parameters **edge** (*Dict*) – An edge

Returns Whether the given edge has passed all defined edge filters

Return type bool

check_node_filter(*node: Dict*) → bool
Check if a node passes defined node filters.

Parameters **node** (*Dict*) – A node

Returns Whether the given node has passed all defined node filters

Return type bool

clear_graph_metadata()
Clears a Source graph's internal graph_metadata. The value of such graph metadata is (now) generally a Callable function. This operation can be used in the code when the metadata is no longer needed, but may cause peculiar Python object persistent problems downstream.

get_infores_catalog() → Dict[str, str]
Return the InfoRes Context of the source

parse(*filename: str*, *format: str*, *compression: Optional[str] = None*, ***kwargs: Any*) → Generator
This method reads from a TSV/CSV and yields records.

Parameters

- **filename** (*str*) – The filename to parse
- **format** (*str*) – The format (`tsv`, `csv`)
- **compression** (*Optional[str]*) – The compression type (`tar`, `tar.gz`)
- **kwargs** (*Any*) – Any additional arguments

Returns A generator for node and edge records

Return type Generator

read_edge (*edge: Dict*) → Optional[Tuple]
Load an edge into an instance of BaseGraph.

Parameters **edge** (*Dict*) – An edge

Returns A tuple that contains subject id, object id, edge key, and edge data

Return type Optional[Tuple]

read_edges (*df: pandas.core.frame.DataFrame*) → Generator
Load edges from pandas.DataFrame into an instance of BaseGraph.

Parameters **df** (*pandas.DataFrame*) – Dataframe containing records that represent edges

Returns A generator for edge records

Return type Generator

read_node (*node: Dict*) → Optional[Tuple[str, Dict]]
Prepare a node.

Parameters **node** (*Dict*) – A node

Returns A tuple that contains node id and node data

Return type Optional[Tuple[str, Dict]]

read_nodes (*df: pandas.core.frame.DataFrame*) → Generator
Read records from pandas.DataFrame and yield records.

Parameters **df** (*pandas.DataFrame*) – Dataframe containing records that represent nodes

Returns A generator for node records

Return type Generator

set_edge_filter (*key: str*, *value: set*) → None
Set an edge filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'subject_category' or 'object_category' filter, the value should be of type `set`. This method also sets the 'category' node filter, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for edge filter
- **value** (*Union[str, set]*) – The value for the edge filter. Can be either a string or a set.

set_edge_filters (*filters: Dict*) → None
Set edge filters.

Parameters **filters** (*Dict*) – Edge filters

set_edge_provenance(*edge_data*)
Set a specific edge provenance value.

set_node_filter(*key: str, value: Union[str, set]*) → None
Set a node filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'category' filter, the value should be of type `set`. This method also sets the 'subject_category' and 'object_category' edge filters, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for node filter
- **value** (*Union[str, set]*) – The value for the node filter. Can be either a string or a set.

set_node_filters(*filters: Dict*) → None
Set node filters.

Parameters **filters** (*Dict*) – Node filters

set_node_provenance(*node_data*)
Set a specific node provenance value.

set_prefix_map(*m: Dict*) → None
Add or override default prefix to IRI map.

Parameters **m** (*Dict*) – Prefix to IRI map

set_provenance_map(*kwargs*)
Set up a provenance (Knowledge Source to InfoRes) map

set_reverse_prefix_map(*m: Dict*) → None
Add or override default IRI to prefix map.

Parameters **m** (*Dict*) – IRI to prefix map

kgx.source.json_source

`JsonSource` is responsible for reading data from a KGX formatted JSON using the ijson library, which allows for streaming data from the file.

class kgx.source.json_source.**JsonSource**
Bases: *kgx.source.tsv_source.TsvSource*

JsonSource is responsible for reading data as records from a JSON.

check_edge_filter(*edge: Dict*) → bool
Check if an edge passes defined edge filters.

Parameters **edge** (*Dict*) – An edge

Returns Whether the given edge has passed all defined edge filters

Return type bool

check_node_filter(*node: Dict*) → bool
Check if a node passes defined node filters.

Parameters **node** (*Dict*) – A node

Returns Whether the given node has passed all defined node filters

Return type bool

clear_graph_metadata()
Clears a Source graph's internal graph_metadata. The value of such graph metadata is (now) generally a Callable function. This operation can be used in the code when the metadata is no longer needed, but may cause peculiar Python object persistent problems downstream.

get_infores_catalog() → Dict[str, str]
Return the InfoRes Context of the source

parse(*filename: str*, *format: str = 'json'*, *compression: Optional[str] = None*, ***kwargs: Any*) → Generator
This method reads from a JSON and yields records.

Parameters

- **filename** (*str*) – The filename to parse
- **format** (*str*) – The format (json)
- **compression** (*Optional[str]*) – The compression type (gz)
- **kwargs** (*Any*) – Any additional arguments

Returns A generator for node and edge records read from the file

Return type Generator

read_edge(*edge: Dict*) → Optional[Tuple]
Load an edge into an instance of BaseGraph.

Parameters **edge** (*Dict*) – An edge

Returns A tuple that contains subject id, object id, edge key, and edge data

Return type Optional[Tuple]

read_edges(*filename: str*) → Generator
Read edge records from a JSON.

Parameters **filename** (*str*) – The filename to read from

Returns A generator for edge records

Return type Generator

read_node(*node: Dict*) → Optional[Tuple[str, Dict]]
Prepare a node.

Parameters **node** (*Dict*) – A node

Returns A tuple that contains node id and node data

Return type Optional[Tuple[str, Dict]]

read_nodes(*filename: str*) → Generator
Read node records from a JSON.

Parameters **filename** (*str*) – The filename to read from

Returns A generator for node records

Return type Generator

set_edge_filter (*key: str*, *value: set*) → None
Set an edge filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'subject_category' or 'object_category' filter, the value should be of type `set`. This method also sets the 'category' node filter, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for edge filter
- **value** (*Union[str, set]*) – The value for the edge filter. Can be either a string or a set.

set_edge_filters (*filters: Dict*) → None
Set edge filters.

Parameters **filters** (*Dict*) – Edge filters

set_edge_provenance (*edge_data*)
Set a specific edge provenance value.

set_node_filter (*key: str, value: Union[str, set]*) → None
Set a node filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'category' filter, the value should be of type `set`. This method also sets the 'subject_category' and 'object_category' edge filters, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for node filter
- **value** (*Union[str, set]*) – The value for the node filter. Can be either a string or a set.

set_node_filters (*filters: Dict*) → None
Set node filters.

Parameters **filters** (*Dict*) – Node filters

set_node_provenance (*node_data*)
Set a specific node provenance value.

set_prefix_map (*m: Dict*) → None
Add or override default prefix to IRI map.

Parameters **m** (*Dict*) – Prefix to IRI map

set_provenance_map (*kwargs*)
Set up a provenance (Knowledge Source to InfoRes) map

set_reverse_prefix_map (*m: Dict*) → None
Add or override default IRI to prefix map.

Parameters **m** (*Dict*) – IRI to prefix map

kgx.source.jsonl_source

`JsonlSource` is responsible for reading data from a KGX formatted JSON Lines using the jsonlines library.

KGX expects two separate JSON Lines files - one for nodes and another for edges.

class kgx.source.jsonl_source.**JsonlSource**
Bases: *kgx.source.json_source.JsonSource*

JsonlSource is responsible for reading data as records from JSON Lines.

check_edge_filter(*edge: Dict*) → bool
Check if an edge passes defined edge filters.

Parameters **edge** (*Dict*) – An edge

Returns Whether the given edge has passed all defined edge filters

Return type bool

check_node_filter(*node: Dict*) → bool
Check if a node passes defined node filters.

Parameters **node** (*Dict*) – A node

Returns Whether the given node has passed all defined node filters

Return type bool

clear_graph_metadata()
Clears a Source graph's internal graph_metadata. The value of such graph metadata is (now) generally a Callable function. This operation can be used in the code when the metadata is no longer needed, but may cause peculiar Python object persistent problems downstream.

get_infores_catalog() → Dict[str, str]
Return the InfoRes Context of the source

parse(*filename: str*, *format: str = 'jsonl'*, *compression: Optional[str] = None*, ***kwargs: Any*) → Generator
This method reads from JSON Lines and yields records.

Parameters

- **filename** (*str*) – The filename to parse
- **format** (*str*) – The format (`json`)
- **compression** (*Optional[str]*) – The compression type (`gz`)
- **kwargs** (*Any*) – Any additional arguments

Returns A generator for records

Return type Generator

read_edge(*edge: Dict*) → Optional[Tuple]
Load an edge into an instance of BaseGraph.

Parameters **edge** (*Dict*) – An edge

Returns A tuple that contains subject id, object id, edge key, and edge data

Return type Optional[Tuple]

read_edges(*filename: str*) → Generator
Read edge records from a JSON.

Parameters **filename** (*str*) – The filename to read from

Returns A generator for edge records

Return type Generator

read_node (*node: Dict*) → Optional[Tuple[str, Dict]]
Prepare a node.

Parameters **node** (*Dict*) – A node

Returns A tuple that contains node id and node data

Return type Optional[Tuple[str, Dict]]

read_nodes (*filename: str*) → Generator
Read node records from a JSON.

Parameters **filename** (*str*) – The filename to read from

Returns A generator for node records

Return type Generator

set_edge_filter (*key: str*, *value: set*) → None
Set an edge filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'subject_category' or 'object_category' filter, the value should be of type `set`. This method also sets the 'category' node filter, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for edge filter
- **value** (*Union[str, set]*) – The value for the edge filter. Can be either a string or a set.

set_edge_filters (*filters: Dict*) → None
Set edge filters.

Parameters **filters** (*Dict*) – Edge filters

set_edge_provenance (*edge_data*)
Set a specific edge provenance value.

set_node_filter (*key: str, value: Union[str, set]*) → None
Set a node filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'category' filter, the value should be of type `set`. This method also sets the 'subject_category' and 'object_category' edge filters, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for node filter
- **value** (*Union[str, set]*) – The value for the node filter. Can be either a string or a set.

set_node_filters (*filters: Dict*) → None
Set node filters.

Parameters **filters** (*Dict*) – Node filters

set_node_provenance (*node_data*)
Set a specific node provenance value.

set_prefix_map (*m: Dict*) → None
Add or override default prefix to IRI map.

Parameters **m** (*Dict*) – Prefix to IRI map

set_provenance_map (*kwargs*)
Set up a provenance (Knowledge Source to InfoRes) map

set_reverse_prefix_map (*m: Dict*) → None
Add or override default IRI to prefix map.

Parameters **m** (*Dict*) – IRI to prefix map

kgx.source.trapi_source

`TrapiSource` is responsible for reading data from a Translator Reasoner API formatted JSON.

class kgx.source.trapi_source.**TrapiSource**
Bases: *kgx.source.json_source.JsonSource*

TrapiSource is responsible for reading data as records from a TRAPI JSON.

check_edge_filter (*edge: Dict*) → bool
Check if an edge passes defined edge filters.

Parameters **edge** (*Dict*) – An edge

Returns Whether the given edge has passed all defined edge filters

Return type bool

check_node_filter (*node: Dict*) → bool
Check if a node passes defined node filters.

Parameters **node** (*Dict*) – A node

Returns Whether the given node has passed all defined node filters

Return type bool

clear_graph_metadata ()
Clears a Source graph's internal graph_metadata. The value of such graph metadata is (now) generally a Callable function. This operation can be used in the code when the metadata is no longer needed, but may cause peculiar Python object persistent problems downstream.

get_infores_catalog () → Dict[str, str]
Return the InfoRes Context of the source

load_edge (*edge: Dict*) → Tuple[str, str, str, Dict]
Load an edge into an instance of BaseGraph

Note: This methods transformers Reasoner Std API format fields to Biolink Model fields.

Parameters **edge** (*Dict*) – An edge

load_node (*node: Dict*) → Tuple[str, Dict]
Load a node into an instance of BaseGraph

Note: This method transformers Reasoner Std API format fields to Biolink Model fields.

Parameters **node** (*Dict*) – A node

parse (*filename: str*, *format: str = 'json'*, *compression: Optional[str] = None*, ***kwargs: Any*) → Generator
This method reads from a JSON and yields records.

Parameters

- **filename** (*str*) – The filename to parse
- **format** (*str*) – The format (`trapi-json`)
- **compression** (*Optional[str]*) – The compression type (`gz`)
- **kwargs** (*Any*) – Any additional arguments

Returns A generator for node and edge records

Return type Generator

read_edge (*edge: Dict*) → Optional[Tuple]
Load an edge into an instance of BaseGraph.

Parameters **edge** (*Dict*) – An edge

Returns A tuple that contains subject id, object id, edge key, and edge data

Return type Optional[Tuple]

read_edges (*filename: str*, *compression: Optional[str] = None*) → Generator
Read edge records from a JSON.

Parameters

- **filename** (*str*) – The filename to read from
- **compression** (*Optional[str]*) – The compression type

Returns A generator for edge records

Return type Generator

read_node (*node: Dict*) → Optional[Tuple[str, Dict]]
Prepare a node.

Parameters **node** (*Dict*) – A node

Returns A tuple that contains node id and node data

Return type Optional[Tuple[str, Dict]]

read_nodes (*filename: str*, *compression: Optional[str] = None*) → Generator
Read node records from a JSON.

Parameters

- **filename** (*str*) – The filename to read from
- **compression** (*Optional[str]*) – The compression type

Returns A generator for node records

Return type Generator

set_edge_filter(*key: str*, *value: set*) → None
Set an edge filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'subject_category' or 'object_category' filter, the value should be of type `set`. This method also sets the 'category' node filter, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for edge filter
- **value** (*Union[str, set]*) – The value for the edge filter. Can be either a string or a set.

set_edge_filters(*filters: Dict*) → None
Set edge filters.

Parameters **filters** (*Dict*) – Edge filters

set_edge_provenance(*edge_data*)
Set a specific edge provenance value.

set_node_filter(*key: str, value: Union[str, set]*) → None
Set a node filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'category' filter, the value should be of type `set`. This method also sets the 'subject_category' and 'object_category' edge filters, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for node filter
- **value** (*Union[str, set]*) – The value for the node filter. Can be either a string or a set.

set_node_filters(*filters: Dict*) → None
Set node filters.

Parameters **filters** (*Dict*) – Node filters

set_node_provenance(*node_data*)
Set a specific node provenance value.

set_prefix_map(*m: Dict*) → None
Add or override default prefix to IRI map.

Parameters **m** (*Dict*) – Prefix to IRI map

set_provenance_map(*kwargs*)
Set up a provenance (Knowledge Source to InfoRes) map

set_reverse_prefix_map(*m: Dict*) → None
Add or override default IRI to prefix map.

Parameters **m** (*Dict*) – IRI to prefix map

kgx.source.obograph_source

`ObographSource` is responsible for reading data from OBOGraphs in JSON.

class kgx.source.obograph_source.**ObographSource**
Bases: *kgx.source.json_source.JsonSource*

ObographSource is responsible for reading data as records from an OBO Graph JSON.

check_edge_filter(*edge: Dict*) → bool
Check if an edge passes defined edge filters.

Parameters **edge** (*Dict*) – An edge

Returns Whether the given edge has passed all defined edge filters

Return type bool

check_node_filter(*node: Dict*) → bool
Check if a node passes defined node filters.

Parameters **node** (*Dict*) – A node

Returns Whether the given node has passed all defined node filters

Return type bool

clear_graph_metadata()
Clears a Source graph's internal graph_metadata. The value of such graph metadata is (now) generally a Callable function. This operation can be used in the code when the metadata is no longer needed, but may cause peculiar Python object persistent problems downstream.

get_category(*curie: str*, *node: dict*) → Optional[str]
Get category for a given CURIE.

Parameters

- **curie** (*str*) – Curie for node
- **node** (*dict*) – Node data

Returns Category for the given node CURIE.

Return type Optional[str]

get_infores_catalog() → Dict[str, str]
Return the InfoRes Context of the source

parse(*filename: str*, *format: str = 'json'*, *compression: Optional[str] = None*, ***kwargs: Any*) → Generator
This method reads from JSON and yields records.

Parameters

- **filename** (*str*) – The filename to parse
- **format** (*str*) – The format (`json`)
- **compression** (*Optional[str]*) – The compression type (`gz`)
- **kwargs** (*Any*) – Any additional arguments

Returns A generator for records

Return type Generator

parse_meta(*node: str*, *meta: Dict*) → Dict
Parse 'meta' field of a node.

Parameters

- **node** (*str*) – Node identifier
- **meta** (*Dict*) – meta dictionary for the node

Returns A dictionary that contains 'description', 'synonyms', 'xrefs', and 'equivalent_nodes'.

Return type Dict

read_edge (*edge: Dict*) → Dict
Read and parse an edge record.

Parameters **edge** (*Dict*) – The edge record

Returns The processed edge

Return type Dict

read_edges (*filename: str*, *compression: Optional[str] = None*) → Generator
Read edge records from a JSON.

Parameters

- **filename** (*str*) – The filename to read from
- **compression** (*Optional[str]*) – The compression type

Returns A generator for edge records

Return type Generator

read_node (*node: Dict*) → Dict
Read and parse a node record.

Parameters **node** (*Dict*) – The node record

Returns The processed node

Return type Dict

read_nodes (*filename: str*, *compression: Optional[str] = None*) → Generator
Read node records from a JSON.

Parameters

- **filename** (*str*) – The filename to read from
- **compression** (*Optional[str]*) – The compression type

Returns A generator for node records

Return type Generator

set_edge_filter (*key: str*, *value: set*) → None
Set an edge filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'subject_category' or 'object_category' filter, the value should be of type `set`. This method also sets the 'category' node filter, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for edge filter

- **value** (*Union[str, set]*) – The value for the edge filter. Can be either a string or a set.

set_edge_filters (*filters: Dict*) → None
Set edge filters.

Parameters **filters** (*Dict*) – Edge filters

set_edge_provenance (*edge_data*)
Set a specific edge provenance value.

set_node_filter (*key: str, value: Union[str, set]*) → None
Set a node filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'category' filter, the value should be of type `set`. This method also sets the 'subject_category' and 'object_category' edge filters, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for node filter
- **value** (*Union[str, set]*) – The value for the node filter. Can be either a string or a set.

set_node_filters (*filters: Dict*) → None
Set node filters.

Parameters **filters** (*Dict*) – Node filters

set_node_provenance (*node_data*)
Set a specific node provenance value.

set_prefix_map (*m: Dict*) → None
Add or override default prefix to IRI map.

Parameters **m** (*Dict*) – Prefix to IRI map

set_provenance_map (*kwargs*)
Set up a provenance (Knowledge Source to InfoRes) map

set_reverse_prefix_map (*m: Dict*) → None
Add or override default IRI to prefix map.

Parameters **m** (*Dict*) – IRI to prefix map

kgx.source.sssom_source

`SssomSource` is responsible for reading data from an SSSOM formatted files.

KGX Source for Simple Standard for Sharing Ontology Mappings ("SSSOM")

class kgx.source.sssom_source.**SssomSource**
Bases: *kgx.source.source.Source*

SssomSource is responsible for reading data as records from an SSSOM file.

check_edge_filter (*edge: Dict*) → bool
Check if an edge passes defined edge filters.

Parameters **edge** (*Dict*) – An edge

Returns Whether the given edge has passed all defined edge filters

Return type bool

check_node_filter(*node: Dict*) → bool
Check if a node passes defined node filters.

Parameters **node** (*Dict*) – A node

Returns Whether the given node has passed all defined node filters

Return type bool

clear_graph_metadata()
Clears a Source graph's internal graph_metadata. The value of such graph metadata is (now) generally a Callable function. This operation can be used in the code when the metadata is no longer needed, but may cause peculiar Python object persistent problems downstream.

get_infores_catalog() → Dict[str, str]
Return the InfoRes Context of the source

load_edge(*edge: Dict*) → Generator
Load an edge into an instance of BaseGraph

Parameters **edge** (*Dict*) – An edge

Returns A generator for node and edge records

Return type Generator

load_edges(*df: pandas.core.frame.DataFrame*) → Generator
Load edges from pandas.DataFrame into an instance of BaseGraph

Parameters **df** (*pandas.DataFrame*) – Dataframe containing records that represent edges

Returns A generator for edge records

Return type Generator

load_node(*node: Dict*) → Tuple[str, Dict]
Load a node into an instance of BaseGraph

Parameters **node** (*Dict*) – A node

Returns A tuple that contains node id and node data

Return type Optional[Tuple[str, Dict]]

parse(*filename: str*, *format: str*, *compression: Optional[str] = None*, ***kwargs: Any*) → Generator
Parse a SSSOM TSV

Parameters

- **filename** (*str*) – File to read from
- **format** (*str*) – The input file format (`tsv`, by default)
- **compression** (*Optional[str]*) – The compression (`gz`)
- **kwargs** (*Dict*) – Any additional arguments

Returns A generator for node and edge records

Return type Generator

parse_header (*filename: str*, *compression: Optional[str] = None*) → None
Parse metadata from SSSOM headers.

Parameters

- **filename** (*str*) – Filename to parse
- **compression** (*Optional[str]*) – Compression type

set_edge_filter (*key: str*, *value: set*) → None
Set an edge filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'subject_category' or 'object_category' filter, the value should be of type `set`. This method also sets the 'category' node filter, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for edge filter
- **value** (*Union[str, set]*) – The value for the edge filter. Can be either a string or a set.

set_edge_filters (*filters: Dict*) → None
Set edge filters.

Parameters **filters** (*Dict*) – Edge filters

set_edge_provenance (*edge_data*)
Set a specific edge provenance value.

set_node_filter (*key: str, value: Union[str, set]*) → None
Set a node filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'category' filter, the value should be of type `set`. This method also sets the 'subject_category' and 'object_category' edge filters, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for node filter
- **value** (*Union[str, set]*) – The value for the node filter. Can be either a string or a set.

set_node_filters (*filters: Dict*) → None
Set node filters.

Parameters **filters** (*Dict*) – Node filters

set_node_provenance (*node_data*)
Set a specific node provenance value.

set_prefix_map (*m: Dict*) → None
Add or override default prefix to IRI map.

Parameters **m** (*Dict*) – Prefix to IRI map

set_provenance_map(*kwargs*)
Set up a provenance (Knowledge Source to InfoRes) map

set_reverse_prefix_map(*m: Dict*) → None
Add or override default IRI to prefix map.

Parameters **m** (*Dict*) – IRI to prefix map

kgx.source.neo_source

`NeoSource` is responsible for reading data from a local or remote Neo4j instance.

class kgx.source.neo_source.**NeoSource**
Bases: *kgx.source.source.Source*

NeoSource is responsible for reading data as records from a Neo4j instance.

check_edge_filter(*edge: Dict*) → bool
Check if an edge passes defined edge filters.

Parameters **edge** (*Dict*) – An edge

Returns Whether the given edge has passed all defined edge filters

Return type bool

check_node_filter(*node: Dict*) → bool
Check if a node passes defined node filters.

Parameters **node** (*Dict*) – A node

Returns Whether the given node has passed all defined node filters

Return type bool

clear_graph_metadata()
Clears a Source graph's internal graph_metadata. The value of such graph metadata is (now) generally a Callable function. This operation can be used in the code when the metadata is no longer needed, but may cause peculiar Python object persistent problems downstream.

count(*is_directed: bool = True*) → int
Get the total count of records to be fetched from the Neo4j database.

Parameters **is_directed** (*bool*) – Are edges directed or undirected. `True`, by default, since edges in most cases are directed.

Returns The total count of records

Return type int

static format_edge_filter(*edge_filters: Dict*, *key: str*, *variable: Optional[str] = None*, *prefix: Optional[str] = None*, *op: Optional[str] = None*) → str
Get the value for edge filter as defined by `key`. This is used as a convenience method for generating cypher queries.

Parameters

- **edge_filters** (*Dict*) – All edge filters
- **key** (*str*) – Name of the edge filter
- **variable** (*Optional[str]*) – Variable binding for cypher query
- **prefix** (*Optional[str]*) – Prefix for the cypher

- **op** (*Optional[str]*) – The operator

Returns Value corresponding to the given edge filter `key`, formatted for CQL

Return type str

static format_node_filter(*node_filters: Dict*, *key: str*, *variable: Optional[str] = None*, *prefix: Optional[str] = None*, *op: Optional[str] = None*) → str

Get the value for node filter as defined by `key`. This is used as a convenience method for generating cypher queries.

Parameters

- **node_filters** (*Dict*) – All node filters
- **key** (*str*) – Name of the node filter
- **variable** (*Optional[str]*) – Variable binding for cypher query
- **prefix** (*Optional[str]*) – Prefix for the cypher
- **op** (*Optional[str]*) – The operator

Returns Value corresponding to the given node filter `key`, formatted for CQL

Return type str

get_edges(*skip: int = 0*, *limit: int = 0*, *is_directed: bool = True*, ***kwargs: Any*) → List

Get a page of edges from the Neo4j database.

Parameters

- **skip** (*int*) – Records to skip
- **limit** (*int*) – Total number of records to query for
- **is_directed** (*bool*) – Are edges directed or undirected (`True`, by default, since edges in most cases are directed)
- **kwargs** (*Any*) – Any additional arguments

Returns A list of 3-tuples

Return type List

get_infores_catalog() → Dict[str, str]

Return the InfoRes Context of the source

get_nodes(*skip: int = 0*, *limit: int = 0*, ***kwargs: Any*) → List

Get a page of nodes from the Neo4j database.

Parameters

- **skip** (*int*) – Records to skip
- **limit** (*int*) – Total number of records to query for
- **kwargs** (*Any*) – Any additional arguments

Returns A list of nodes

Return type List

get_pages(*query_function*, *start: int = 0*, *end: Optional[int] = None*, *page_size: int = 50000*, ***kwargs: Any*) → Iterator

Get pages of size `page_size` from Neo4j. Returns an iterator of pages where number of pages is (`end - start`)/`page_size`

Parameters

- **query_function** (*func*) – The function to use to fetch records. Usually this is `self.get_nodes` or `self.get_edges`
- **start** (*int*) – Start for pagination
- **end** (*Optional[int]*) – End for pagination
- **page_size** (*int*) – Size of each page (`10000`, by default)
- **kwargs** (*Dict*) – Any additional arguments that might be relevant for `query_function`

Returns An iterator for a list of records from Neo4j. The size of the list is `page_size`

Return type Iterator

load_edge (*edge_record: List*) → Tuple
Load an edge into an instance of BaseGraph

Parameters **edge_record** (*List*) – A 4-tuple edge record

Returns A tuple with subject ID, object ID, edge key, and edge data

Return type Tuple

load_edges (*edges: List*) → None
Load edges into an instance of BaseGraph

Parameters **edges** (*List*) – A list of edge records

load_node (*node: Dict*) → Tuple
Load node into an instance of BaseGraph

Parameters **node** (*Dict*) – A node

Returns A tuple with node ID and node data

Return type Tuple

load_nodes (*nodes: List*) → None
Load nodes into an instance of BaseGraph

Parameters **nodes** (*List*) – A list of nodes

parse (*uri: str*, *username: str*, *password: str*, *node_filters: Dict = None*, *edge_filters: Dict = None*, *start: int = 0*, *end: int = None*, *is_directed: bool = True*, *page_size: int = 50000*, ***kwargs: Any*) → Generator
This method reads from Neo4j instance and yields records

Parameters

- **uri** (*str*) – The URI for the Neo4j instance. For example, http://localhost:7474
- **username** (*str*) – The username
- **password** (*str*) – The password
- **node_filters** (*Dict*) – Node filters
- **edge_filters** (*Dict*) – Edge filters
- **start** (*int*) – Number of records to skip before streaming
- **end** (*int*) – Total number of records to fetch
- **is_directed** (*bool*) – Whether or not the edges should be treated as directed
- **page_size** (*int*) – The size of each page/batch fetched from Neo4j (`50000`)

- **kwargs** (*Any*) – Any additional arguments

Returns A generator for records

Return type Generator

set_edge_filter(*key: str*, *value: set*) → None
Set an edge filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'subject_category' or 'object_category' filter, the value should be of type `set`. This method also sets the 'category' node filter, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for edge filter
- **value** (*Union[str, set]*) – The value for the edge filter. Can be either a string or a set.

set_edge_filters(*filters: Dict*) → None
Set edge filters.

Parameters **filters** (*Dict*) – Edge filters

set_edge_provenance(*edge_data*)
Set a specific edge provenance value.

set_node_filter(*key: str, value: Union[str, set]*) → None
Set a node filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'category' filter, the value should be of type `set`. This method also sets the 'subject_category' and 'object_category' edge filters, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for node filter
- **value** (*Union[str, set]*) – The value for the node filter. Can be either a string or a set.

set_node_filters(*filters: Dict*) → None
Set node filters.

Parameters **filters** (*Dict*) – Node filters

set_node_provenance(*node_data*)
Set a specific node provenance value.

set_prefix_map(*m: Dict*) → None
Update default prefix map.

Parameters **m** (*Dict*) – A dictionary with prefix to IRI mappings

set_provenance_map(*kwargs*)
Set up a provenance (Knowledge Source to InfoRes) map

kgx.source.rdf_source

`RdfSource` is responsible for reading data from RDF N-Triples.

This source makes use of a custom `kgx.parsers.ntriples_parser.CustomNTriplesParser` for parsing N-Triples, which extends `rdflib.plugins.parsers.ntriples.NTriplesParser`.

To ensure proper parsing of N-Triples and a relatively low memory footprint, it is recommended that the N-Triples be sorted based on the subject IRIs.

```
sort -k 1,2 -t ' ' data.nt > data_sorted.nt
```

class kgx.source.rdf_source.**RdfSource**
Bases: *kgx.source.source.Source*

RdfSource is responsible for reading data as records from RDF.

Note: Currently only RDF N-Triples are supported.

---

add_edge(*subject_iri: rdflib.term.URIRef*, *object_iri: rdflib.term.URIRef*, *predicate_iri: rdflib.term.URIRef*, *data: Optional[Dict[Any, Any]] = None*) → Dict
Add an edge to cache.

Parameters

- **subject_iri** (*rdflib.URIRef*) – Subject IRI for the subject in a triple
- **object_iri** (*rdflib.URIRef*) – Object IRI for the object in a triple
- **predicate_iri** (*rdflib.URIRef*) – Predicate IRI for the predicate in a triple
- **data** (*Optional[Dict[Any, Any]]*) – Additional edge properties

Returns The edge data

Return type Dict

add_node(*iri: rdflib.term.URIRef*, *data: Optional[Dict] = None*) → Dict
Add a node to cache.

Parameters

- **iri** (*rdflib.URIRef*) – IRI of a node
- **data** (*Optional[Dict]*) – Additional node properties

Returns The node data

Return type Dict

add_node_attribute(*iri: Union[rdflib.term.URIRef, str], key: str, value: Union[str, List]*) → None
Add an attribute to a node in cache, while taking into account whether the attribute should be multi-valued.

The `key` may be a rdflib.URIRef or an URI string that maps onto a property name as defined in `rdf_utils.property_mapping`.

Parameters

- **iri** (*Union[rdflib.URIRef, str]*) – The IRI of a node in the rdflib.Graph
- **key** (*str*) – The name of the attribute. Can be a rdflib.URIRef or URI string
- **value** (*Union[str, List]*) – The value of the attribute

Returns The node data

Return type Dict

check_edge_filter(*edge: Dict*) → bool
Check if an edge passes defined edge filters.

Parameters **edge** (*Dict*) – An edge

Returns Whether the given edge has passed all defined edge filters

Return type bool

check_node_filter(*node: Dict*) → bool
Check if a node passes defined node filters.

Parameters **node** (*Dict*) – A node

Returns Whether the given node has passed all defined node filters

Return type bool

clear_graph_metadata()
Clears a Source graph's internal graph_metadata. The value of such graph metadata is (now) generally a Callable function. This operation can be used in the code when the metadata is no longer needed, but may cause peculiar Python object persistent problems downstream.

dereify(*n: str*, *node: Dict*) → None
Dereify a node to create a corresponding edge.

Parameters

- **n** (*str*) – Node identifier
- **node** (*Dict*) – Node data

get_biolink_element(*predicate: Any*) → Optional[linkml_runtime.linkml_model.meta.Element]
Returns a Biolink Model element for a given predicate.

Parameters **predicate** (*Any*) – The CURIE of a predicate

Returns The corresponding Biolink Model element

Return type Optional[Element]

get_infores_catalog() → Dict[str, str]
Return the InfoRes Context of the source

parse(*filename: str*, *format: str = 'nt'*, *compression: Optional[str] = None*, ***kwargs: Any*) → Generator
This method reads from RDF N-Triples and yields records.

Note: To ensure proper parsing of N-Triples and a relatively low memory footprint, it is recommended that the N-Triples be sorted based on the subject IRIs.

```
`sort -k 1,2 -t ' ' data.nt > data_sorted.nt`
```

Parameters

- **filename** (*str*) – The filename to parse
- **format** (*str*) – The format (nt)
- **compression** (*Optional[str]*) – The compression type (gz)
- **kwargs** (*Any*) – Any additional arguments

Returns A generator for records

Return type Generator

process_predicate(*p: Union[rdflib.term.URIRef, str, None]*) → Tuple
Process a predicate where the method checks if there is a mapping in Biolink Model.

Parameters **p** (*Optional[Union[URIRef, str]]*) – The predicate

Returns A tuple that contains the Biolink CURIE (if available), the Biolink slot_uri CURIE (if available), the CURIE form of p, the reference of p

Return type Tuple

set_edge_filter(*key: str*, *value: set*) → None
Set an edge filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'subject_category' or 'object_category' filter, the value should be of type `set`. This method also sets the 'category' node filter, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for edge filter
- **value** (*Union[str, set]*) – The value for the edge filter. Can be either a string or a set.

set_edge_filters(*filters: Dict*) → None
Set edge filters.

Parameters **filters** (*Dict*) – Edge filters

set_edge_provenance(*edge_data*)
Set a specific edge provenance value.

set_node_filter(*key: str, value: Union[str, set]*) → None
Set a node filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'category' filter, the value should be of type `set`. This method also sets the 'subject_category' and 'object_category' edge filters, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for node filter
- **value** (*Union[str, set]*) – The value for the node filter. Can be either a string or a set.

set_node_filters(*filters: Dict*) → None
Set node filters.

Parameters **filters** (*Dict*) – Node filters

set_node_property_predicates(*predicates*) → None
Set predicates that are to be treated as node properties.

Parameters **predicates** (*Set*) – Set of predicates

set_node_provenance(*node_data*)
Set a specific node provenance value.

set_predicate_mapping(*m: Dict*) → None
Set predicate mappings.

Use this method to update mappings for predicates that are not in Biolink Model.

Parameters **m** (*Dict*) – A dictionary where the keys are IRIs and values are their corresponding property names

set_prefix_map(*m: Dict*) → None
Update default prefix map.

Parameters **m** (*Dict*) – A dictionary with prefix to IRI mappings

set_provenance_map(*kwargs*)
Set up a provenance (Knowledge Source to InfoRes) map

triple(*s: rdflib.term.URIRef*, *p: rdflib.term.URIRef*, *o: rdflib.term.URIRef*) → None
Parse a triple.

Parameters

- **s** (*URIRef*) – Subject
- **p** (*URIRef*) – Predicate
- **o** (*URIRef*) – Object

update_edge(*subject_curie: str, object_curie: str, edge_key: str, data: Optional[Dict[Any, Any]]*) → Dict
Update an edge with properties.

Parameters

- **subject_curie** (*str*) – Subject CURIE
- **object_curie** (*str*) – Object CURIE
- **edge_key** (*str*) – Edge key
- **data** (*Optional[Dict[Any, Any]]*) – Edge properties

Returns The edge data

Return type Dict

update_node(*n: Union[rdflib.term.URIRef, str], data: Optional[Dict] = None*) → Dict
Update a node with properties.

Parameters

- **n** (*Union[URIRef, str]*) – Node identifier
- **data** (*Optional[Dict]*) – Node properties

Returns The node data

Return type Dict

kgx.source.owl_source

`OwlSource` is responsible for parsing an OWL ontology.

When parsing an OWL, this source also adds OwlStar annotations to certain OWL axioms.

class kgx.source.owl_source.**OwlSource**
Bases: *kgx.source.rdf_source.RdfSource*

OwlSource is responsible for parsing an OWL ontology.

..note:: This is a simple parser that loads direct class-class relationships. For more formal OWL parsing, refer to Robot: http://robot.obolibrary.org/

add_edge(*subject_iri: rdflib.term.URIRef*, *object_iri: rdflib.term.URIRef*, *predicate_iri: rdflib.term.URIRef*, *data: Optional[Dict[Any, Any]] = None*) → Dict
Add an edge to cache.

Parameters

- **subject_iri** (*rdflib.URIRef*) – Subject IRI for the subject in a triple
- **object_iri** (*rdflib.URIRef*) – Object IRI for the object in a triple
- **predicate_iri** (*rdflib.URIRef*) – Predicate IRI for the predicate in a triple
- **data** (*Optional[Dict[Any, Any]]*) – Additional edge properties

Returns The edge data

Return type Dict

add_node(*iri: rdflib.term.URIRef*, *data: Optional[Dict] = None*) → Dict
Add a node to cache.

Parameters

- **iri** (*rdflib.URIRef*) – IRI of a node
- **data** (*Optional[Dict]*) – Additional node properties

Returns The node data

Return type Dict

add_node_attribute(*iri: Union[rdflib.term.URIRef, str], key: str, value: Union[str, List]*) → None
Add an attribute to a node in cache, while taking into account whether the attribute should be multi-valued.

The `key` may be a rdflib.URIRef or an URI string that maps onto a property name as defined in `rdf_utils.property_mapping`.

Parameters

- **iri** (*Union[rdflib.URIRef, str]*) – The IRI of a node in the rdflib.Graph
- **key** (*str*) – The name of the attribute. Can be a rdflib.URIRef or URI string
- **value** (*Union[str, List]*) – The value of the attribute

Returns The node data

Return type Dict

check_edge_filter(*edge: Dict*) → bool
Check if an edge passes defined edge filters.

Parameters **edge** (*Dict*) – An edge

Returns Whether the given edge has passed all defined edge filters

Return type bool

check_node_filter(*node: Dict*) → bool
Check if a node passes defined node filters.

Parameters **node** (*Dict*) – A node

Returns Whether the given node has passed all defined node filters

Return type bool

clear_graph_metadata()
Clears a Source graph's internal graph_metadata. The value of such graph metadata is (now) generally a Callable function. This operation can be used in the code when the metadata is no longer needed, but may cause peculiar Python object persistent problems downstream.

dereify(*n: str*, *node: Dict*) → None
Dereify a node to create a corresponding edge.

Parameters

- **n** (*str*) – Node identifier
- **node** (*Dict*) – Node data

get_biolink_element(*predicate: Any*) → Optional[linkml_runtime.linkml_model.meta.Element]
Returns a Biolink Model element for a given predicate.

Parameters **predicate** (*Any*) – The CURIE of a predicate

Returns The corresponding Biolink Model element

Return type Optional[Element]

get_infores_catalog() → Dict[str, str]
Return the InfoRes Context of the source

load_graph(*rdfgraph: rdflib.graph.Graph*, ***kwargs: Any*) → None
Walk through the rdflib.Graph and load all triples into kgx.graph.base_graph.BaseGraph

Parameters

- **rdfgraph** (*rdflib.Graph*) – Graph containing nodes and edges
- **kwargs** (*Any*) – Any additional arguments

parse(*filename: str*, *format: str = 'owl'*, *compression: Optional[str] = None*, ***kwargs: Any*) → Generator
This method reads from an OWL and yields records.

Parameters

- **filename** (*str*) – The filename to parse
- **format** (*str*) – The format (owl)
- **compression** (*Optional[str]*) – The compression type (gz)
- **kwargs** (*Any*) – Any additional arguments

Returns A generator for node and edge records read from the file

Return type Generator

process_predicate(*p: Union[rdflib.term.URIRef, str, None]*) → Tuple
Process a predicate where the method checks if there is a mapping in Biolink Model.

Parameters **p** (*Optional[Union[URIRef, str]]*) – The predicate

Returns A tuple that contains the Biolink CURIE (if available), the Biolink slot_uri CURIE (if available), the CURIE form of p, the reference of p

Return type Tuple

set_edge_filter (*key: str*, *value: set*) → None
Set an edge filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'subject_category' or 'object_category' filter, the value should be of type `set`. This method also sets the 'category' node filter, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for edge filter
- **value** (*Union[str, set]*) – The value for the edge filter. Can be either a string or a set.

set_edge_filters (*filters: Dict*) → None
Set edge filters.

Parameters **filters** (*Dict*) – Edge filters

set_edge_provenance (*edge_data*)
Set a specific edge provenance value.

set_node_filter (*key: str, value: Union[str, set]*) → None
Set a node filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'category' filter, the value should be of type `set`. This method also sets the 'subject_category' and 'object_category' edge filters, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for node filter
- **value** (*Union[str, set]*) – The value for the node filter. Can be either a string or a set.

set_node_filters (*filters: Dict*) → None
Set node filters.

Parameters **filters** (*Dict*) – Node filters

set_node_property_predicates (*predicates*) → None
Set predicates that are to be treated as node properties.

Parameters **predicates** (*Set*) – Set of predicates

set_node_provenance (*node_data*)
Set a specific node provenance value.

set_predicate_mapping (*m: Dict*) → None
Set predicate mappings.

Use this method to update mappings for predicates that are not in Biolink Model.

Parameters **m** (*Dict*) – A dictionary where the keys are IRIs and values are their corresponding property names

set_prefix_map (*m: Dict*) → None
Update default prefix map.

Parameters **m** (*Dict*) – A dictionary with prefix to IRI mappings

set_provenance_map (*kwargs*)
Set up a provenance (Knowledge Source to InfoRes) map

triple (*s: rdflib.term.URIRef*, *p: rdflib.term.URIRef*, *o: rdflib.term.URIRef*) → None
Parse a triple.

Parameters

- **s** (*URIRef*) – Subject
- **p** (*URIRef*) – Predicate
- **o** (*URIRef*) – Object

update_edge (*subject_curie: str, object_curie: str, edge_key: str, data: Optional[Dict[Any, Any]]*) → Dict
Update an edge with properties.

Parameters

- **subject_curie** (*str*) – Subject CURIE
- **object_curie** (*str*) – Object CURIE
- **edge_key** (*str*) – Edge key
- **data** (*Optional[Dict[Any, Any]]*) – Edge properties

Returns The edge data

Return type Dict

update_node (*n: Union[rdflib.term.URIRef, str], data: Optional[Dict] = None*) → Dict
Update a node with properties.

Parameters

- **n** (*Union[URIRef, str]*) – Node identifier
- **data** (*Optional[Dict]*) – Node properties

Returns The node data

Return type Dict

kgx.source.sparql_source

`SparqlSource` has yet to be implemented.

In principle, `SparqlSource` should be able to read data from a local or remote SPARQL endpoint.

class kgx.source.sparql_source.**SparqlSource**
Bases: *kgx.source.rdf_source.RdfSource*

add_edge (*subject_iri: rdflib.term.URIRef*, *object_iri: rdflib.term.URIRef*, *predicate_iri: rdflib.term.URIRef*, *data: Optional[Dict[Any, Any]] = None*) → Dict
Add an edge to cache.

Parameters

- **subject_iri** (*rdflib.URIRef*) – Subject IRI for the subject in a triple
- **object_iri** (*rdflib.URIRef*) – Object IRI for the object in a triple
- **predicate_iri** (*rdflib.URIRef*) – Predicate IRI for the predicate in a triple
- **data** (*Optional[Dict[Any, Any]]*) – Additional edge properties

Returns The edge data

Return type Dict

add_node(*iri: rdflib.term.URIRef*, *data: Optional[Dict] = None*) → Dict
Add a node to cache.

Parameters

- **iri** (*rdflib.URIRef*) – IRI of a node
- **data** (*Optional[Dict]*) – Additional node properties

Returns The node data

Return type Dict

add_node_attribute(*iri: Union[rdflib.term.URIRef, str], key: str, value: Union[str, List]*) → None
Add an attribute to a node in cache, while taking into account whether the attribute should be multi-valued.

The `key` may be a rdflib.URIRef or an URI string that maps onto a property name as defined in `rdf_utils.property_mapping`.

Parameters

- **iri** (*Union[rdflib.URIRef, str]*) – The IRI of a node in the rdflib.Graph
- **key** (*str*) – The name of the attribute. Can be a rdflib.URIRef or URI string
- **value** (*Union[str, List]*) – The value of the attribute

Returns The node data

Return type Dict

check_edge_filter(*edge: Dict*) → bool
Check if an edge passes defined edge filters.

Parameters **edge** (*Dict*) – An edge

Returns Whether the given edge has passed all defined edge filters

Return type bool

check_node_filter(*node: Dict*) → bool
Check if a node passes defined node filters.

Parameters **node** (*Dict*) – A node

Returns Whether the given node has passed all defined node filters

Return type bool

clear_graph_metadata()
Clears a Source graph's internal graph_metadata. The value of such graph metadata is (now) generally a Callable function. This operation can be used in the code when the metadata is no longer needed, but may cause peculiar Python object persistent problems downstream.

dereify(*n: str*, *node: Dict*) → None
Dereify a node to create a corresponding edge.

Parameters

- **n** (*str*) – Node identifier
- **node** (*Dict*) – Node data

get_biolink_element(*predicate: Any*) → Optional[linkml_runtime.linkml_model.meta.Element]
Returns a Biolink Model element for a given predicate.

Parameters **predicate** (*Any*) – The CURIE of a predicate

Returns The corresponding Biolink Model element

Return type Optional[Element]

get_infores_catalog() → Dict[str, str]
Return the InfoRes Context of the source

parse(*filename: str*, *format: str = 'nt'*, *compression: Optional[str] = None*, ***kwargs: Any*) → Generator
This method reads from RDF N-Triples and yields records.

Note: To ensure proper parsing of N-Triples and a relatively low memory footprint, it is recommended that the N-Triples be sorted based on the subject IRIs.

```
`sort -k 1,2 -t ' ' data.nt > data_sorted.nt`
```

Parameters

- **filename** (*str*) – The filename to parse
- **format** (*str*) – The format (`nt`)
- **compression** (*Optional[str]*) – The compression type (`gz`)
- **kwargs** (*Any*) – Any additional arguments

Returns A generator for records

Return type Generator

process_predicate(*p: Union[rdflib.term.URIRef, str, None]*) → Tuple
Process a predicate where the method checks if there is a mapping in Biolink Model.

Parameters **p** (*Optional[Union[URIRef, str]]*) – The predicate

Returns A tuple that contains the Biolink CURIE (if available), the Biolink slot_uri CURIE (if available), the CURIE form of p, the reference of p

Return type Tuple

set_edge_filter(*key: str*, *value: set*) → None
Set an edge filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'subject_category' or 'object_category' filter, the value should be of type `set`. This method also sets the 'category' node filter, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for edge filter

- **value** (*Union[str, set]*) – The value for the edge filter. Can be either a string or a set.

set_edge_filters(*filters: Dict*) → None
Set edge filters.

Parameters **filters** (*Dict*) – Edge filters

set_edge_provenance(*edge_data*)
Set a specific edge provenance value.

set_node_filter(*key: str, value: Union[str, set]*) → None
Set a node filter, as defined by a key and value pair. These filters are used to filter (or reduce) the search space when fetching nodes from the underlying store.

Note: When defining the 'category' filter, the value should be of type `set`. This method also sets the 'subject_category' and 'object_category' edge filters, to get a consistent set of nodes in the subgraph.

Parameters

- **key** (*str*) – The key for node filter
- **value** (*Union[str, set]*) – The value for the node filter. Can be either a string or a set.

set_node_filters(*filters: Dict*) → None
Set node filters.

Parameters **filters** (*Dict*) – Node filters

set_node_property_predicates(*predicates*) → None
Set predicates that are to be treated as node properties.

Parameters **predicates** (*Set*) – Set of predicates

set_node_provenance(*node_data*)
Set a specific node provenance value.

set_predicate_mapping(*m: Dict*) → None
Set predicate mappings.

Use this method to update mappings for predicates that are not in Biolink Model.

Parameters **m** (*Dict*) – A dictionary where the keys are IRIs and values are their corresponding property names

set_prefix_map(*m: Dict*) → None
Update default prefix map.

Parameters **m** (*Dict*) – A dictionary with prefix to IRI mappings

set_provenance_map(*kwargs*)
Set up a provenance (Knowledge Source to InfoRes) map

triple(*s: rdflib.term.URIRef, p: rdflib.term.URIRef, o: rdflib.term.URIRef*) → None
Parse a triple.

Parameters

- **s** (*URIRef*) – Subject
- **p** (*URIRef*) – Predicate

- **o** (*URIRef*) – Object

update_edge(*subject_curie: str, object_curie: str, edge_key: str, data: Optional[Dict[Any, Any]]*) → Dict

Update an edge with properties.

Parameters

- **subject_curie** (*str*) – Subject CURIE
- **object_curie** (*str*) – Object CURIE
- **edge_key** (*str*) – Edge key
- **data** (*Optional[Dict[Any, Any]]*) – Edge properties

Returns The edge data

Return type Dict

update_node(*n: Union[rdflib.term.URIRef, str], data: Optional[Dict] = None*) → Dict

Update a node with properties.

Parameters

- **n** (*Union[URIRef, str]*) – Node identifier
- **data** (*Optional[Dict]*) – Node properties

Returns The node data

Return type Dict

1.2.5 Sink

A Sink can be implemented for any file, local, and/or remote store to which a graph can be written to. A Sink is responsible for writing nodes and edges from a graph.

A Sink must subclass `kgx.sink.sink.Sink` class and must implement the following methods:

- `__init__`
- `write_nodes`
- `write_edges`
- `finalize`

`__init__` method

The `__init__` method is used to instantiate a Sink with configurations required for writing to a store.

- In the case of files, the `__init__` method will take the `filename` and `format` as arguments
- In the case of a graph store like Neo4j, the `__init__` method will take the `uri`, `username`, and `password` as arguments.

The `__init__` method also has an optional `kwargs` argument which can be used to supply variable number of arguments to this method, depending on the requirements for the store for which the Sink is being implemented.

`write_nodes` method

- Responsible for receiving a node record and writing to a file/store

`write_edges` method

- Responsible for receiving an edge record and writing to a file/store

`finalize` method

Any operation that needs to be performed after writing all the nodes and edges to a file/store must be defined in this method.

For example,

- `kgx.source.tsv_source.TsvSource` has a `finalize` method that closes the file handles and creates an archive, if compression is desired
- `kgx.source.neo_sink.NeoSink` has a `finalize` method that writes any cached node and edge records

kgx.sink.sink

Base class for all Sinks in KGX.

class kgx.sink.sink.**Sink**
Bases: `object`

A Sink is responsible for writing data as records to a store where the store is a file or a database.

finalize() → None
Operations that ought to be done after writing all the incoming data should be called by this method.

set_reverse_prefix_map(*m: Dict*) → None
Update default reverse prefix map.

Parameters **m** (*Dict*) – A dictionary with IRI to prefix mappings

write_edge(*record*) → None
Write an edge record to the underlying store.

Parameters **record** (*Any*) – An edge record

write_node(*record*) → None
Write a node record to the underlying store.

Parameters **record** (*Any*) – A node record

kgx.sink.graph_sink

`GraphSink` is responsible for writing to an instance of `kgx.graph.base_graph.BaseGraph` and must use only the methods exposed by `BaseGraph` to access the graph.

class kgx.sink.graph_sink.**GraphSink**(*graph: kgx.graph.base_graph.BaseGraph = None*)
Bases: *kgx.sink.sink.Sink*

GraphSink is responsible for writing data as records to an in memory graph representation.

The underlying store is determined by the graph store class defined in config (`kgx.graph.nx_graph.NxGraph`, by default).

Parameters **graph** (kgx.graph.base_graph.BaseGraph) – An instance of BaseGraph to read from

finalize() → None
Perform any operations after writing nodes and edges to graph.

set_reverse_prefix_map(*m: Dict*) → None
Update default reverse prefix map.

Parameters **m** (*Dict*) – A dictionary with IRI to prefix mappings

write_edge(*record: Dict*) → None
Write an edge record to graph.

Parameters **record** (*Dict*) – An edge record

write_node(*record: Dict*) → None
Write a node record to graph.

Parameters **record** (*Dict*) – A node record

kgx.sink.tsv_sink

TsvSink is responsible for writing a KGX formatted CSV or TSV using Pandas.

KGX writes two separate files - one for nodes and another for edges.

class kgx.sink.tsv_sink.**TsvSink**(*filename: str*, *format: str*, *compression: Optional[str] = None*, ***kwargs: Any*)

Bases: *kgx.sink.sink.Sink*

TsvSink is responsible for writing data as records to a TSV/CSV.

Parameters

- **filename** (*str*) – The filename to write to
- **format** (*str*) – The file format (tsv, csv)
- **compression** (*str*) – The compression type (tar, tar.gz)
- **kwargs** (*Any*) – Any additional arguments

finalize() → None
Close file handles and create an archive if compression mode is defined.

set_edge_properties(*edge_properties: List*) → None
Update edge properties index with a given list.

Parameters **edge_properties** (*List*) – A list of edge properties

set_node_properties(*node_properties: List*) → None
Update node properties index with a given list.

Parameters **node_properties** (*List*) – A list of node properties

set_reverse_prefix_map(*m: Dict*) → None
Update default reverse prefix map.

Parameters **m** (*Dict*) – A dictionary with IRI to prefix mappings

write_edge(*record: Dict*) → None
Write an edge record to the underlying store.

Parameters **record** (*Dict*) – An edge record

write_node (*record: Dict*) → None
Write a node record to the underlying store.

Parameters **record** (*Dict*) – A node record

kgx.sink.json_sink

`JsonSink` is responsible for writing a KGX formatted JSON using the jsonstreams library, which allows for streaming records to the file.

class kgx.sink.json_sink.**JsonSink** (*filename: str, format: str = 'json', compression: Optional[str] = None, **kwargs: Any*)
Bases: *kgx.sink.sink.Sink*

JsonSink is responsible for writing data as records to a JSON.

Parameters

- **filename** (*str*) – The filename to write to
- **format** (*str*) – The file format (`json`)
- **compression** (*Optional[str]*) – The compression type (`gz`)
- **kwargs** (*Any*) – Any additional arguments

finalize () → None
Finalize by creating a compressed file, if needed.

set_reverse_prefix_map (*m: Dict*) → None
Update default reverse prefix map.

Parameters **m** (*Dict*) – A dictionary with IRI to prefix mappings

write_edge (*record: Dict*) → None
Write an edge record to JSON.

Parameters **record** (*Dict*) – An edge record

write_node (*record: Dict*) → None
Write a node record to JSON.

Parameters **record** (*Dict*) – A node record

kgx.sink.jsonl_sink

`JsonlSink` is responsible for writing a KGX formatted JSON Lines using the jsonlines library.

KGX writes two separate JSON Lines files - one for nodes and another for edges.

class kgx.sink.jsonl_sink.**JsonlSink** (*filename: str, format: str = 'jsonl', compression: Optional[str] = None, **kwargs: Any*)
Bases: *kgx.sink.sink.Sink*

JsonlSink is responsible for writing data as records to JSON lines.

Parameters

- **filename** (*str*) – The filename to write to
- **format** (*str*) – The file format (`jsonl`)
- **compression** (*Optional[str]*) – The compression type (`gz`)

- **kwargs** (*Any*) – Any additional arguments

finalize() → None
Perform any operations after writing the file.

set_reverse_prefix_map(*m: Dict*) → None
Update default reverse prefix map.

Parameters **m** (*Dict*) – A dictionary with IRI to prefix mappings

write_edge(*record: Dict*) → None
Write an edge record to JSON.

Parameters **record** (*Dict*) – A node record

write_node(*record: Dict*) → None
Write a node record to JSON.

Parameters **record** (*Dict*) – A node record

kgx.sink.trapi_sink

`TrapiSink` has yet to be implemented.

In principle, `TrapiSink` is responsible for writing a Translator Reasoner API formatted JSON.

class kgx.sink.trapi_sink.**TrapiSink**(*filename: str*, *format: str*, *compression: Optional[str] = None*, ***kwargs: Any*)
Bases: *kgx.sink.tsv_sink.TsvSink*

finalize() → None
Close file handles and create an archive if compression mode is defined.

set_edge_properties(*edge_properties: List*) → None
Update edge properties index with a given list.

Parameters **edge_properties** (*List*) – A list of edge properties

set_node_properties(*node_properties: List*) → None
Update node properties index with a given list.

Parameters **node_properties** (*List*) – A list of node properties

set_reverse_prefix_map(*m: Dict*) → None
Update default reverse prefix map.

Parameters **m** (*Dict*) – A dictionary with IRI to prefix mappings

write_edge(*record: Dict*) → None
Write an edge record to the underlying store.

Parameters **record** (*Dict*) – An edge record

write_node(*record: Dict*) → None
Write a node record to the underlying store.

Parameters **record** (*Dict*) – A node record

kgx.sink.neo_sink

`NeoSink` is responsible for writing data to a local or remote Neo4j instance.

class kgx.sink.neo_sink.**NeoSink**(*uri: str*, *username: str*, *password: str*, ***kwargs: Any*)
Bases: *kgx.sink.sink.Sink*

NeoSink is responsible for writing data as records to a Neo4j instance.

Parameters

- **uri** (*str*) – The URI for the Neo4j instance. For example, http://localhost:7474
- **username** (*str*) – The username
- **password** (*str*) – The password
- **kwargs** (*Any*) – Any additional arguments

static create_constraint_query(*category: str*) → str
Create a Cypher CONSTRAINT query

Parameters **category** (*str*) – The category to create a constraint on

Returns The Cypher CONSTRAINT query

Return type str

create_constraints(*categories: Union[set, list]*) → None
Create a unique constraint on node 'id' for all `categories` in Neo4j.

Parameters **categories** (*Union[set, list]*) – Set of categories

finalize() → None
Write any remaining cached node and/or edge records.

static generate_unwind_edge_query(*edge_predicate: str*) → str
Generate UNWIND cypher query for saving edges into Neo4j.

Query uses `self.DEFAULT_NODE_CATEGORY` to quickly lookup the required subject and object node.

Parameters **edge_predicate** (*str*) – Edge label as string

Returns The UNWIND cypher query

Return type str

static generate_unwind_node_query(*category: str*) → str
Generate UNWIND cypher query for saving nodes into Neo4j.

There should be a CONSTRAINT in Neo4j for `self.DEFAULT_NODE_CATEGORY`. The query uses `self.DEFAULT_NODE_CATEGORY` as the node label to increase speed for adding nodes. The query also sets label to `self.DEFAULT_NODE_CATEGORY` for any node to make sure that the CONSTRAINT applies.

Parameters **category** (*str*) – Node category

Returns The UNWIND cypher query

Return type str

static sanitize_category(*category: List*) → List
Sanitize category for use in UNWIND cypher clause. This method adds escape characters to each element in category list to ensure the category is processed correctly.

Parameters **category** (*List*) – Category

Returns Sanitized category list

Return type List

set_reverse_prefix_map(*m: Dict*) → None
Update default reverse prefix map.

Parameters **m** (*Dict*) – A dictionary with IRI to prefix mappings

write_edge(*record*) → None
Cache an edge record that is to be written to Neo4j. This method writes a cache of edge records when the total number of records exceeds CACHE_SIZE

Parameters **record** (*Dict*) – An edge record

write_node(*record*) → None
Cache a node record that is to be written to Neo4j. This method writes a cache of node records when the total number of records exceeds CACHE_SIZE

Parameters **record** (*Dict*) – A node record

kgx.sink.rdf_sink

RdfSink is responsible for writing data as RDF N-Triples.

class kgx.sink.rdf_sink.**RdfSink**(*filename: str*, *format: str = 'nt'*, *compression: Optional[bool] = None*, *reify_all_edges: bool = False*, ***kwargs: Any*)
Bases: *kgx.sink.sink.Sink*

RdfSink is responsible for writing data as records to an RDF serialization.

Note: Currently only RDF N-Triples serialization is supported.

Parameters

- **filename** (*str*) – The filename to write to
- **format** (*str*) – The file format (nt)
- **compression** (*str*) – The compression type (gz)
- **reify_all_edges** (*bool*) – Whether or not to reify all the edges
- **kwargs** (*Any*) – Any additional arguments

finalize() → None
Perform any operations after writing the file.

get_biolink_element(*predicate: Any*) → Optional[linkml_runtime.linkml_model.meta.Element]
Returns a Biolink Model element for a given predicate.

Parameters **predicate** (*Any*) – The CURIE of a predicate

Returns The corresponding Biolink Model element

Return type Optional[Element]

process_predicate(*p: Union[rdflib.term.URIRef, str, None]*) → Tuple
Process a predicate where the method checks if there is a mapping in Biolink Model.

Parameters **p** (*Optional[Union[URIRef, str]]*) – The predicate

Returns A tuple that contains the Biolink CURIE (if available), the Biolink slot_uri CURIE (if available), the CURIE form of p, the reference of p

Return type Tuple

reify(*u: str*, *v: str*, *data: Dict*) → Dict
Create a node representation of an edge.

Parameters

- **u** (*str*) – Subject
- **v** (*str*) – Object
- **k** (*str*) – Edge key
- **data** (*Dict*) – Edge data

Returns The reified node

Return type Dict

set_property_types(*m: Dict*) → None
Set export type for properties that are not in Biolink Model.

Parameters **m** (*Dict*) – A dictionary where the keys are property names and values are their corresponding types.

set_reverse_predicate_mapping(*m: Dict*) → None
Set reverse predicate mappings.

Use this method to update mappings for predicates that are not in Biolink Model.

Parameters **m** (*Dict*) – A dictionary where the keys are property names and values are their corresponding IRI.

set_reverse_prefix_map(*m: Dict*) → None
Update default reverse prefix map.

Parameters **m** (*Dict*) – A dictionary with IRI to prefix mappings

uriref(*identifier: str*) → rdflib.term.URIRef
Generate a rdflib.URIRef for a given string.

Parameters **identifier** (*str*) – Identifier as string.

Returns URIRef form of the input `identifier`

Return type rdflib.URIRef

write_edge(*record: Dict*) → None
Write an edge record as triples.

Parameters **record** (*Dict*) – An edge record

write_node(*record: Dict*) → None
Write a node record as triples.

Parameters **record** (*Dict*) – A node record

1.2.6 Validator

The Validator validates an instance of kgx.graph.base_graph.BaseGraph for Biolink Model compliance.

To validate a graph,

```
from kgx.validator import Validator
v = Validator()
v.validate(graph)
```

Streaming Data Processing Mode

For very large graphs, the Validator operation may now successfully process graph data equally well using data streaming (command flag `--stream=True`) which significantly minimizes the memory footprint required to process such graphs.

Biolink Model Versioning

By default, the Validator validates against the latest Biolink Model release hosted by the current Biolink Model Toolkit; hwoever, one may override this default at the Validator class level using the `Validator.set_biolink_model(version="#.#.#")` where **#.#.#** is the *major.minor.patch* semantic versioning of the desired Biolink Model release.

Every instance of Validator() persistently assumes the most recently set class level Biolink Model version. Resetting the class level Biolink Model does not change the version of previously instantiated Validator() objects. In a multi-threaded environment instantiating multiple validator objects, it may be necessary to wrap the `Validator.set_biolink_model` and `Validator()` object instantiation together within a single thread locked block.

Note that the kgx **validate** CLI operation also has an optional `biolink_release` argument for the same purpose.

kgx.validator

class kgx.validator.**ErrorType**
Bases: enum.Enum

Validation error types

class kgx.validator.**MessageLevel**
Bases: enum.Enum

Message level for validation reports

class kgx.validator.**ValidationError**(*entity: str*, *error_type: kgx.validator.ErrorType*, *message: str*, *message_level: kgx.validator.MessageLevel*)
Bases: object

ValidationError class that represents an error.

Parameters

- **entity** (*str*) – The node or edge entity that is failing validation
- **error_type** (kgx.validator.ErrorType) – The nature of the error
- **message** (*str*) – The error message
- **message_level** (kgx.validator.MessageLevel) – The message level

class kgx.validator.**Validator**(*verbose: bool = False, progress_monitor: Optional[Callable[[kgx.utils.kgx_utils.GraphEntityType, List], None]] = None, schema: Optional[str] = None*)

Bases: `object`

Class for validating a property graph.

The optional 'progress_monitor' for the validator should be a lightweight Callable which is injected into the class 'inspector' Callable, designed to intercepts node and edge records streaming through the Validator (inside a Transformer.process() call. The first (GraphEntityType) argument of the Callable tags the record as a NODE or an EDGE. The second argument given to the Callable is the current record itself. This Callable is strictly meant to be procedural and should *not* mutate the record. The intent of this Callable is to provide a hook to KGX applications wanting the namesake function of passively monitoring the graph data stream. As such, the Callable could simply tally up the number of times it is called with a NODE or an EDGE, then provide a suitable (quick!) report of that count back to the KGX application. The Callable (function/callable class) should not modify the record and should be of low complexity, so as not to introduce a large computational overhead to validation!

Parameters

- **verbose** (*bool*) – Whether the generated report should be verbose or not (default: `False`)
- **progress_monitor** (*Optional[Callable[[GraphEntityType, List], None]]*) – Function given a peek at the current record being processed by the class wrapped Callable.
- **schema** (*Optional[str]*) – URL to (Biolink) Model Schema to be used for validated (default: None, use default Biolink Model Toolkit schema)

__call__(*entity_type: kgx.utils.kgx_utils.GraphEntityType*, *rec: List*)
Transformer 'inspector' Callable

static get_all_prefixes(*jsonld: Optional[Dict] = None*) → set
Get all prefixes from Biolink Model JSON-LD context.

It also sets `self.prefixes` for subsequent access.

Parameters **jsonld** (*Optional[Dict]*) – The JSON-LD context

Returns A set of prefixes

Return type Optional[Dict]

get_error_messages()
A direct Validator "instance" method version of report() that directly accesses the internal Validator self.errors list.

Returns A list of formatted error messages.

Return type List

static get_required_edge_properties(*toolkit: Optional[bmt.toolkit.Toolkit] = None*) → list
Get all properties for an edge that are required, as defined by Biolink Model.

Parameters **toolkit** (*Optional[Toolkit]*) – Optional externally provided toolkit (default: use Validator class defined toolkit)

Returns A list of required edge properties

Return type list

static get_required_node_properties(*toolkit: Optional[bmt.toolkit.Toolkit] = None*) → list

Get all properties for a node that are required, as defined by Biolink Model.

Parameters **toolkit** (*Optional[Toolkit]*) – Optional externally provided toolkit (default: use Validator class defined toolkit)

Returns A list of required node properties

Return type list

static report(*errors: List[kgx.validator.ValidationError]*) → List

Prepare error report.

Parameters **errors** (*List[ValidationError]*) – List of kgx.validator.ValidationError

Returns A list of formatted errors

Return type List

validate(*graph: kgx.graph.base_graph.BaseGraph*) → list

Validate nodes and edges in a graph. TODO: Support strict mode

Parameters **graph** (kgx.graph.base_graph.BaseGraph) – The graph to validate

Returns A list of errors for a given graph

Return type list

static validate_categories(*node: str, data: dict, toolkit: Optional[bmt.toolkit.Toolkit] = None*) → list

Validate `category` field of a given node.

Parameters

- **node** (*str*) – Node identifier
- **data** (*dict*) – Node properties
- **toolkit** (*Optional[Toolkit]*) – Optional externally provided toolkit (default: use Validator class defined toolkit)

Returns A list of errors for a given node

Return type list

static validate_edge_predicate(*subject: str, object: str, data: dict, toolkit: Optional[bmt.toolkit.Toolkit] = None*) → list

Validate `edge_predicate` field of a given edge.

Parameters

- **subject** (*str*) – Subject identifier
- **object** (*str*) – Object identifier
- **data** (*dict*) – Edge properties
- **toolkit** (*Optional[Toolkit]*) – Optional externally provided toolkit (default: use Validator class defined toolkit)

Returns A list of errors for a given edge

Return type list

static validate_edge_properties(*subject: str, object: str, data: dict, required_properties: list*) → list

Checks if all the required edge properties exist for a given edge.

Parameters

- **subject** (*str*) – Subject identifier
- **object** (*str*) – Object identifier
- **data** (*dict*) – Edge properties
- **required_properties** (*list*) – Required edge properties

Returns A list of errors for a given edge

Return type list

static validate_edge_property_types(*subject: str*, *object: str*, *data: dict*, *toolkit: Optional[bmt.toolkit.Toolkit] = None*) → list

Checks if edge properties have the expected value type.

Parameters

- **subject** (*str*) – Subject identifier
- **object** (*str*) – Object identifier
- **data** (*dict*) – Edge properties
- **toolkit** (*Optional[Toolkit]*) – Optional externally provided toolkit (default: use Validator class defined toolkit)

Returns A list of errors for a given edge

Return type list

static validate_edge_property_values(*subject: str*, *object: str*, *data: dict*) → list

Validate an edge property's value.

Parameters

- **subject** (*str*) – Subject identifier
- **object** (*str*) – Object identifier
- **data** (*dict*) – Edge properties

Returns A list of errors for a given edge

Return type list

validate_edges(*graph: kgx.graph.base_graph.BaseGraph*) → list

Validate all the edges in a graph.

This method validates for the following, - Edge properties - Edge property type - Edge property value type - Edge predicate

Parameters **graph** (kgx.graph.base_graph.BaseGraph) – The graph to validate

Returns A list of errors for a given graph

Return type list

static validate_node_properties(*node: str*, *data: dict*, *required_properties: list*) → list

Checks if all the required node properties exist for a given node.

Parameters

- **node** (*str*) – Node identifier
- **data** (*dict*) – Node properties
- **required_properties** (*list*) – Required node properties

Returns A list of errors for a given node

Return type list

static validate_node_property_types(*node: str*, *data: dict*, *toolkit: Optional[bmt.toolkit.Toolkit] = None*) → list

Checks if node properties have the expected value type.

Parameters

- **node** (*str*) – Node identifier
- **data** (*dict*) – Node properties
- **toolkit** (*Optional[Toolkit]*) – Optional externally provided toolkit (default: use Validator class defined toolkit)

Returns A list of errors for a given node

Return type list

static validate_node_property_values(*node: str*, *data: dict*) → list

Validate a node property's value.

Parameters

- **node** (*str*) – Node identifier
- **data** (*dict*) – Node properties

Returns A list of errors for a given node

Return type list

validate_nodes(*graph: kgx.graph.base_graph.BaseGraph*) → list

Validate all the nodes in a graph.

This method validates for the following, - Node properties - Node property type - Node property value type - Node categories

Parameters **graph** (kgx.graph.base_graph.BaseGraph) – The graph to validate

Returns A list of errors for a given graph

Return type list

write_report(*outstream: TextIO*) → None

Write error report to a file

Parameters **outstream** (*TextIO*) – The stream to write to

1.2.7 Graph Operations

This module provides a set of graph operations that are supported by KGX. Each operation has an entrypoint - a function that takes an instance of kgx.graph.base_graph.BaseGraph as input and performs an operation on the nodes and/or edges of that graph.

Clique Merge

The Clique Merge operation performs a series of operations on your target (input) graph:

- Build cliques from nodes in the target graph
- Elect a leader for each individual clique
- Move all edges in a clique to the leader node

The main entry point is `kgx.graph_operations.clique_merge.clique_merge` method which takes an instance of `kgx.graph.base_graph.BaseGraph`.

Build cliques from nodes in the target graph

Given a target graph, create a clique graph where nodes in the same clique are connected via `biolink:same_as` edges.

In the target graph, you can define nodes that belong to the same clique as follows:

- Having `biolink:same_as` edges between nodes (preferred and consistent with Biolink Model)
- Having `same_as` node property on a node that lists all equivalent nodes (deprecated)

Elect a leader for each individual clique

Once the clique graph is built, go through each clique and elect a representative node or leader node for that clique.

Elect leader for each clique based on three election criteria, listed in the order in which they are checked:

- **Leader annotation:** Elect the leader node for a clique based on `clique_leader` annotation on the node
- **Prefix prioritization:** Elect the leader node for a clique that has a prefix which is of the highest priority in the identifier prefixes list, as defined in the Biolink Model
- **Prefix prioritization fallback:** Elect the leader node for a clique that has a prefix which is the first in an alphabetically sorted list of all ID prefixes within the clique

Move all edges in a clique to the leader node

The last step is edge consolidation where all the edges from nodes in a clique are moved to the leader node.

The original subject and object node of an edge is tracked via the `_original_subject` and `_original_object` edge property.

kgx.graph_operations.clique_merge

`kgx.graph_operations.clique_merge.`**`build_cliques`**(*target_graph: kgx.graph.base_graph.BaseGraph*) → networkx.classes.multidigraph.MultiDiGraph

Builds a clique graph from `same_as` edges in `target_graph`.

Parameters **`target_graph`** (`kgx.graph.base_graph.BaseGraph`) – An instance of BaseGraph that contains nodes and edges

Returns The clique graph with only `same_as` edges

Return type networkx.MultiDiGraph

`kgx.graph_operations.clique_merge.`**`check_all_categories`**(*categories*) → Tuple[List, List, List]

Check all categories in `categories`.

Parameters **`categories`** (*`List`*) – A list of categories

Returns

- *Tuple[List, List, List]* – A tuple consisting of valid biolink categories, invalid biolink categories, and invalid categories
- **Note** (*the sort_categories method will re-arrange the passed in category list according to the distance*)
- *of each list member from the top of their hierarchy. Each category's hierarchy is made up of its*
- *'is_a' and mixin ancestors.*

kgx.graph_operations.clique_merge.**check_categories**(*categories: List*, *closure: List*, *category_mapping: Optional[Dict[str, str]] = None*) → Tuple[List, List, List]

Check categories to ensure whether values in `categories` are valid biolink categories. Valid biolink categories are classes that descend from 'NamedThing'. Mixins, while valid ancestors, are not valid categories.

Parameters

- **categories** (*List*) – A list of categories to check
- **closure** (*List*) – A list of nodes in a clique
- **category_mapping** (*Optional[Dict[str, str]]*) – A map that provides mapping from a non-biolink category to a biolink category

Returns A tuple consisting of valid biolink categories, invalid biolink categories, and invalid categories

Return type Tuple[List, List, List]

kgx.graph_operations.clique_merge.**clique_merge**(*target_graph: kgx.graph.base_graph.BaseGraph*, *leader_annotation: str = None*, *prefix_prioritization_map: Optional[Dict[str, List[str]]] = None*, *category_mapping: Optional[Dict[str, str]] = None*, *strict: bool = True*) → Tuple[kgx.graph.base_graph.BaseGraph, networkx.classes.multidigraph.MultiDiGraph]

Parameters

- **target_graph** (kgx.graph.base_graph.BaseGraph) – The original graph
- **leader_annotation** (*str*) – The field on a node that signifies that the node is the leader of a clique
- **prefix_prioritization_map** (*Optional[Dict[str, List[str]]]*) – A map that gives a prefix priority for one or more categories
- **category_mapping** (*Optional[Dict[str, str]]*) – Mapping for non-Biolink Model categories to Biolink Model categories
- **strict** (*bool*) – Whether or not to merge nodes in a clique that have conflicting node categories

Returns A tuple containing the updated target graph, and the clique graph

Return type Tuple[*kgx.graph.base_graph.BaseGraph*, networkx.MultiDiGraph]

kgx.graph_operations.clique_merge.**consolidate_edges**(*target_graph: kgx.graph.base_graph.BaseGraph*, *clique_graph: networkx.classes.multidigraph.MultiDiGraph*, *leader_annotation: str*) → kgx.graph.base_graph.BaseGraph

Move all edges from nodes in a clique to the clique leader.

Original subject and object of a node are preserved via `ORIGINAL_SUBJECT_PROPERTY` and `ORIGINAL_OBJECT_PROPERTY`

Parameters

- **target_graph** (`kgx.graph.base_graph.BaseGraph`) – The original graph
- **clique_graph** (*networkx.MultiDiGraph*) – The clique graph
- **leader_annotation** (*str*) – The field on a node that signifies that the node is the leader of a clique

Returns The target graph where all edges from nodes in a clique are moved to clique leader

Return type *kgx.graph.base_graph.BaseGraph*

kgx.graph_operations.clique_merge.**elect_leader**(*target_graph: kgx.graph.base_graph.BaseGraph*, *clique_graph: networkx.classes.multidigraph.MultiDiGraph*, *leader_annotation: str*, *prefix_prioritization_map: Optional[Dict[str, List[str]]]*, *category_mapping: Optional[Dict[str, str]]*, *strict: bool = True*) → kgx.graph.base_graph.BaseGraph

Elect leader for each clique in a graph.

Parameters

- **target_graph** (`kgx.graph.base_graph.BaseGraph`) – The original graph
- **clique_graph** (*networkx.Graph*) – The clique graph
- **leader_annotation** (*str*) – The field on a node that signifies that the node is the leader of a clique
- **prefix_prioritization_map** (*Optional[Dict[str, List[str]]]*) – A map that gives a prefix priority for one or more categories
- **category_mapping** (*Optional[Dict[str, str]]*) – Mapping for non-Biolink Model categories to Biolink Model categories
- **strict** (*bool*) – Whether or not to merge nodes in a clique that have conflicting node categories

Returns The updated target graph

Return type *kgx.graph.base_graph.BaseGraph*

kgx.graph_operations.clique_merge.**get_category_from_equivalence**(*target_graph: kgx.graph.base_graph.BaseGraph*, *clique_graph: networkx.classes.multidigraph.MultiDiGra*, *node: str*, *attributes: Dict*) → List

Get category for a node based on its equivalent nodes in a graph.

Parameters

- **target_graph** (kgx.graph.base_graph.BaseGraph) – The original graph
- **clique_graph** (*networkx.MultiDiGraph*) – The clique graph
- **node** (*str*) – Node identifier
- **attributes** (*Dict*) – Node's attributes

Returns Category for the node

Return type List

kgx.graph_operations.clique_merge.**get_clique_category**(*clique_graph: networkx.classes.multidigraph.MultiDiGraph*, *clique: List*) → Tuple[str, List]

Given a clique, identify the category of the clique.

Parameters

- **clique_graph** (*nx.MultiDiGraph*) – Clique graph
- **clique** (*List*) – A list of nodes in clique

Returns A tuple of clique category and its ancestors

Return type Tuple[str, list]

kgx.graph_operations.clique_merge.**get_leader_by_annotation**(*target_graph: kgx.graph.base_graph.BaseGraph*, *clique_graph: networkx.classes.multidigraph.MultiDiGraph*, *clique: List*, *leader_annotation: str*) → Tuple[Optional[str], Optional[str]]

Get leader by searching for leader annotation property in any of the nodes in a given clique.

Parameters

- **target_graph** (kgx.graph.base_graph.BaseGraph) – The original graph
- **clique_graph** (*networkx.MultiDiGraph*) – The clique graph
- **clique** (*List*) – A list of nodes from a clique
- **leader_annotation** (*str*) – The field on a node that signifies that the node is the leader of a clique

Returns A tuple containing the node that has been elected as the leader and the election strategy

Return type Tuple[Optional[str], Optional[str]]

kgx.graph_operations.clique_merge.**get_leader_by_prefix_priority**(*target_graph: kgx.graph.base_graph.BaseGraph*, *clique_graph: networkx.classes.multidigraph.MultiDiGra*, *clique: List*, *prefix_priority_list: List*) → Tuple[Optional[str], Optional[str]]

Get leader from clique based on a given prefix priority.

Parameters

- **target_graph** (kgx.graph.base_graph.BaseGraph) – The original graph
- **clique_graph** (*networkx.MultiDiGraph*) – The clique graph
- **clique** (*List*) – A list of nodes that correspond to a clique
- **prefix_priority_list** (*List*) – A list of prefixes in descending priority

Returns A tuple containing the node that has been elected as the leader and the election strategy

Return type Tuple[Optional[str], Optional[str]]

kgx.graph_operations.clique_merge.**get_leader_by_sort**(*target_graph: kgx.graph.base_graph.BaseGraph*, *clique_graph: networkx.classes.multidigraph.MultiDiGraph*, *clique: List*) → Tuple[Optional[str], Optional[str]]

Get leader from clique based on the first selection from an alphabetical sort of the node id prefixes.

Parameters

- **target_graph** (kgx.graph.base_graph.BaseGraph) – The original graph
- **clique_graph** (*networkx.MultiDiGraph*) – The clique graph
- **clique** (*List*) – A list of nodes that correspond to a clique

Returns A tuple containing the node that has been elected as the leader and the election strategy

Return type Tuple[Optional[str], Optional[str]]

kgx.graph_operations.clique_merge.**sort_categories**(*categories: Union[List, Set, ordered_set.OrderedSet]*) → List

Sort a list of categories from most specific to the most generic.

Parameters **categories** (*Union[List, Set, OrderedSet]*) – A list of categories

Returns A sorted list of categories where sorted means that the first element in the list returned has the most number of parents in the class hierarchy.

Return type List

kgx.graph_operations.clique_merge.**update_node_categories**(*target_graph: kgx.graph.base_graph.BaseGraph, clique_graph: networkx.classes.multidigraph.MultiDiGraph, clique: List, category_mapping: Optional[Dict[str, str]], strict: bool = True*) → List

For a given clique, get category for each node in clique and validate against Biolink Model, mapping to Biolink Model category where needed.

For example, If a node has `biolink:Gene` as its category, then this method adds all of its ancestors.

Parameters

- **target_graph** (kgx.graph.base_graph.BaseGraph) – The original graph
- **clique_graph** (*networkx.Graph*) – The clique graph
- **clique** (*List*) – A list of nodes from a clique
- **category_mapping** (*Optional[Dict[str, str]]*) – Mapping for non-Biolink Model categories to Biolink Model categories
- **strict** (*bool*) – Whether or not to merge nodes in a clique that have conflicting node categories

Returns The clique

Return type List

Graph Merge

The Graph Merge operation takes one or more instances of `kgx.graph.base_graph.BaseGraph` and merges them into a single graph.

Depending on the desired outcome, there are two entry points for merging graphs:

- `kgx.graph_operations.graph_merge.merge_all_graphs`: This method takes a list of graphs, identifies the largest graph in the list and merges all the remaining graphs to the largest graph. This is done to reduce the memory footprint. The side-effect is that the incoming graphs are modified during this operation.
- `kgx.graph_operations.graph_merge.merge_graphs`: This method takes a list of graphs and merges all of them into a new graph. While this approach ensures that the incoming graphs are not modified, there is an increased memory requirement to accommodate the newly created graph.

Following are the criteria used for merging graphs:

- Two nodes are said to be identical if they have the same `id`
- If a two identical nodes have conflicting node properties,
  - when `preserve` is `True`, the values for the properties are concatenated to a list, if and only if the node property is not a core node property
  - when `preserve` is `False`, the values for the properties are replaced with the values from the incoming node, if and only if the node property is not a core node property
- Two edges are said to be identical if they have the same `subject`, `object` and edge `key`, where the edge `key` can be a pre-defined UUID or these are IDs autogenerated using and edge's `subject`, `predicate`, and `object`

- If a two identical edges have conflicting edges properties,
  - when `preserve` is `True`, the values for the properties are concatenated to a list, if and only if the edge property is not a core edge property
  - when `preserve` is `False`, the values for the properties are replaced with the values from the incoming edge, if and only if the edge property is not a core edge property

kgx.graph_operations.graph_merge

kgx.graph_operations.graph_merge.**add_all_edges**(*g1: kgx.graph.base_graph.BaseGraph*, *g2: kgx.graph.base_graph.BaseGraph*, *preserve: bool = True*) → int

Add all edges from source graph (`g2`) to target graph (`g1`).

Parameters

- **g1** (kgx.graph.base_graph.BaseGraph) – Target graph
- **g2** (kgx.graph.base_graph.BaseGraph) – Source graph
- **preserve** (*bool*) – Whether or not to preserve conflicting properties

Returns Number of edges merged during this operation

Return type int

kgx.graph_operations.graph_merge.**add_all_nodes**(*g1: kgx.graph.base_graph.BaseGraph*, *g2: kgx.graph.base_graph.BaseGraph*, *preserve: bool = True*) → int

Add all nodes from source graph (`g2`) to target graph (`g1`).

Parameters

- **g1** (kgx.graph.base_graph.BaseGraph) – Target graph
- **g2** (kgx.graph.base_graph.BaseGraph) – Source graph
- **preserve** (*bool*) – Whether or not to preserve conflicting properties

Returns Number of nodes merged during this operation

Return type int

kgx.graph_operations.graph_merge.**merge_all_graphs**(*graphs: List[kgx.graph.base_graph.BaseGraph]*, *preserve: bool = True*) → kgx.graph.base_graph.BaseGraph

Merge one or more graphs.

Note: This method will first pick the largest graph in `graphs` and use that as the target to merge the remaining graphs. This is to reduce the memory footprint for this operation. The criteria for largest graph is the graph with the largest number of edges.

The caveat is that the merge operation has a side effect where the largest graph is altered.

If you would like to ensure that all incoming graphs remain as-is, then look at `merge_graphs`.

The outcome of the merge on node and edge properties depend on the `preserve` parameter. If preserve is `True` then, - core properties will not be overwritten - other properties will be concatenated to a list

If preserve is `False` then, - core properties will not be overwritten - other properties will be replaced

Parameters

- **graphs** (*List[kgx.graph.base_graph.BaseGraph]*) – A list of instances of BaseGraph to merge
- **preserve** (*bool*) – Whether or not to preserve conflicting properties

Returns The merged graph

Return type *kgx.graph.base_graph.BaseGraph*

kgx.graph_operations.graph_merge.**merge_edge**(*g: kgx.graph.base_graph.BaseGraph*, *u: str*, *v: str*, *key: str*, *data: dict*, *preserve: bool = True*) → dict

Merge edge `u` -> `v` into graph `g`.

Parameters

- **g** (kgx.graph.base_graph.BaseGraph) – The target graph
- **u** (*str*) – Subject node id
- **v** (*str*) – Object node id
- **key** (*str*) – Edge key
- **data** (*dict*) – Node properties
- **preserve** (*bool*) – Whether or not to preserve conflicting properties

Returns The merged edge

Return type dict

kgx.graph_operations.graph_merge.**merge_graphs**(*graph: kgx.graph.base_graph.BaseGraph, graphs: List[kgx.graph.base_graph.BaseGraph], preserve: bool = True*) → kgx.graph.base_graph.BaseGraph

Merge all graphs in `graphs` to `graph`.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – An instance of BaseGraph
- **graphs** (*List[kgx.graph.base_graph.BaseGraph]*) – A list of instances of BaseGraph to merge
- **preserve** (*bool*) – Whether or not to preserve conflicting properties

Returns The merged graph

Return type *kgx.graph.base_graph.BaseGraph*

kgx.graph_operations.graph_merge.**merge_node**(*g: kgx.graph.base_graph.BaseGraph*, *n: str*, *data: dict*, *preserve: bool = True*) → dict

Merge node `n` into graph `g`.

Parameters

- **g** (kgx.graph.base_graph.BaseGraph) – The target graph
- **n** (*str*) – Node id
- **data** (*dict*) – Node properties
- **preserve** (*bool*) – Whether or not to preserve conflicting properties

Returns The merged node

Return type dict

Summarize Graph

The Summarize Graph operation takes an instance of `kgx.graph.base_graph.BaseGraph` and generates summary statistics for the entire graph.

This operation generates summary as a YAML (or JSON) in a format that is compatible with the Knowledge Graph Hub dashboard.

The main entry point is the `kgx.graph_operations.summarize_graph.generate_graph_stats` method.

The tool does detect and logs anomalies in the graph (defaults reporting to stderr, but may be reset to a file using the `error_log` parameter)

Note: To generate a summary statistics YAML that is consistent with Translator API (TRAPI) Release 1.1 standards, refer to Meta Knowledge Graph.

Streaming Data Processing Mode

For very large graphs, the Graph Summary operation may now successfully process graph data equally well using data streaming (command flag `--stream=True`) which significantly minimizes the memory footprint required to process such graphs.

kgx.graph_operations.summarize_graph

class kgx.graph_operations.summarize_graph.**GraphSummary**(*name=''*, *node_facet_properties: Optional[List] = None*, *edge_facet_properties: Optional[List] = None*, *progress_monitor: Optional[Callable[[kgx.utils.kgx_utils.GraphEntityTyp List], None]] = None*, *error_log: str = None*, ***kwargs*)

Bases: `object`

Class for generating a "classical" knowledge graph summary.

The optional 'progress_monitor' for the validator should be a lightweight Callable which is injected into the class 'inspector' Callable, designed to intercepts node and edge records streaming through the Validator (inside a Transformer.process() call. The first (GraphEntityType) argument of the Callable tags the record as a NODE or an EDGE. The second argument given to the Callable is the current record itself. This Callable is strictly meant to be procedural and should *not* mutate the record. The intent of this Callable is to provide a hook to KGX applications wanting the namesake function of passively monitoring the graph data stream. As such, the Callable could simply tally up the number of times it is called with a NODE or an EDGE, then provide a suitable (quick!) report of that count back to the KGX application. The Callable (function/callable class) should not modify the record and should be of low complexity, so as not to introduce a large computational overhead to validation!

class Category(*category_curie: str*, *summary*)

Bases: `object`

Internal class for compiling statistics about a distinct category.

__init__ (*category_curie: str*, *summary*)
GraphSummary.Category constructor.
category: str Biolink Model category curie identifier.

analyse_node_category (*summary*, *n*, *data*)
Analyse metadata of a given graph node record of this category.
Parameters
- **summary** (GraphSummary) – GraphSunmmary within which the Category is being analysed.
- **n** (*str*) – Curie identifier of the node record (not used here).
- **data** (*Dict*) – Complete data dictionary of node record fields.

get_cid() → int
Returns Internal GraphSummary index id for tracking a Category.
Return type int

get_count()
Returns Count of nodes which have this category.
Return type int

get_count_by_id_prefixes()
Returns Count of nodes by id_prefixes for nodes which have this category.
Return type int

get_id_prefixes() → Set
Returns Set of identifier prefix (strings) used by nodes of this Category.
Return type Set[str]

get_name() → str
Returns Biolink CURIE name of the category.
Return type str

json_object()
Returns Returns JSON friendly metadata for this category.,
Return type Dict[str, Any]

__call__ (*entity_type: kgx.utils.kgx_utils.GraphEntityType*, *rec: List*)
Transformer 'inspector' Callable, for analysing a stream of graph data.

Parameters

- **entity_type** (GraphEntityType) – indicates what kind of record being passed to the function for analysis.
- **rec** (*Dict*) – Complete data dictionary of the given record.

__init__ (*name=''*, *node_facet_properties: Optional[List] = None*, *edge_facet_properties: Optional[List] = None*, *progress_monitor: Optional[Callable[[kgx.utils.kgx_utils.GraphEntityType, List], None]] = None*, *error_log: str = None*, ***kwargs*)
GraphSummary constructor.

Parameters

- **name** (*str*) – (Graph) name assigned to the summary.
- **node_facet_properties** (*Optional[List]*) – A list of properties to facet on. For example, `['provided_by']`
- **edge_facet_properties** (*Optional[List]*) – A list of properties to facet on. For example, `['knowledge_source']`

- **progress_monitor** (*Optional[Callable[[GraphEntityType, List], None]]*) – Function given a peek at the current record being stream processed by the class wrapped Callable.
- **error_log** (*str*) – Where to write any graph processing error message (stderr, by default)

add_node_stat(*tag: str*, *value: Any*)
Compile/add a nodes statistic for a given tag = value annotation of the node.

Parameters

- **tag** (*str*) –
- **value** (*Any*) –
- **tag** – Tag label for the annotation.
- **value** – Value of the specific tag annotation.

Returns

analyse_edge(*u: str*, *v: str*, *k: str*, *data: Dict*)
Analyse metadata of one graph edge record.

Parameters

- **u** (*str*) – Subject node curie identifier of the edge.
- **v** (*str*) – Subject node curie identifier of the edge.
- **k** (*str*) – Key identifier of the edge record (not used here).
- **data** (*Dict*) – Complete data dictionary of edge record fields.

analyse_node(*n*, *data*)
Analyse metadata of one graph node record.

Parameters

- **n** (*str*) – Curie identifier of the node record (not used here).
- **data** (*Dict*) – Complete data dictionary of node record fields.

get_category(*category_curie: str*) → kgx.graph_operations.summarize_graph.GraphSummary.Category
Counts the number of distinct (Biolink) categories encountered in the knowledge graph (not including those of 'unknown' category)

Parameters **category_curie** (*str*) – Curie identifier for the (Biolink) category.

Returns MetaKnowledgeGraph.Category object for a given Biolink category.

Return type Category

get_facet_counts(*data: Dict*, *stats: Dict*, *x: str*, *y: str*, *facet_property: str*) → Dict
Facet on `facet_property` and record the count for `stats[x][y][facet_property]`.

Parameters

- **data** (*dict*) – Node/edge data dictionary
- **stats** (*dict*) – The stats dictionary
- **x** (*str*) – first key
- **y** (*str*) – second key
- **facet_property** (*str*) – The property to facet on

Returns The stats dictionary

Return type Dict

get_graph_summary (*name: str = None*, ***kwargs*) → Dict
Similar to summarize_graph except that the node and edge statistics are already captured in the GraphSummary class instance (perhaps by Transformer.process() stream inspection) and therefore, the data structure simply needs to be 'finalized' for saving or similar use.

Parameters

- **name** (*Optional[str]*) – Name for the graph (if being renamed)
- **kwargs** (*Dict*) – Any additional arguments (ignored in this method at present)

Returns A knowledge map dictionary corresponding to the graph

Return type Dict

get_name ()

Returns Currently assigned knowledge graph name.

Return type str

get_node_stats () → Dict[str, Any]

Returns Statistics for the nodes in the graph.

Return type Dict[str, Any]

save (*file*, *name: str = None*, *file_format: str = 'yaml'*)
Save the current GraphSummary to a specified (open) file (device).

Parameters

- **file** (*File*) – Text file handler open for writing.
- **name** (*str*) – Optional string to which to (re-)name the graph.
- **file_format** (*str*) – Text output format ('json' or 'yaml') for the saved meta knowledge graph (default: 'json')

Returns

Return type None

summarize_graph (*graph: kgx.graph.base_graph.BaseGraph*) → Dict
Summarize the entire graph.

Parameters **graph** (kgx.graph.base_graph.BaseGraph) – The graph

Returns The stats dictionary

Return type Dict

summarize_graph_edges (*graph: kgx.graph.base_graph.BaseGraph*) → Dict
Summarize the edges in a graph.

Parameters **graph** (kgx.graph.base_graph.BaseGraph) – The graph

Returns The edge stats

Return type Dict

summarize_graph_nodes (*graph: kgx.graph.base_graph.BaseGraph*) → Dict
Summarize the nodes in a graph.

Parameters **graph** (kgx.graph.base_graph.BaseGraph) – The graph

Returns The node stats

Return type Dict

kgx.graph_operations.summarize_graph.**generate_graph_stats**(*graph: kgx.graph.base_graph.BaseGraph*, *graph_name: str*, *filename: str*, *node_facet_properties: Optional[List] = None*, *edge_facet_properties: Optional[List] = None*) → None

Generate stats from Graph.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph
- **graph_name** (*str*) – Name for the graph
- **filename** (*str*) – Filename to write the stats to
- **node_facet_properties** (*Optional[List]*) – A list of properties to facet on. For example, `['provided_by']`
- **edge_facet_properties** (*Optional[List]*) – A list of properties to facet on. For example, `['knowledge_source']`

kgx.graph_operations.summarize_graph.**gs_default**(*o*)

JSONEncoder 'default' function override to properly serialize 'Set' objects (into 'List')

kgx.graph_operations.summarize_graph.**summarize_graph**(*graph: kgx.graph.base_graph.BaseGraph*, *name: str = None*, *node_facet_properties: Optional[List] = None*, *edge_facet_properties: Optional[List] = None*) → Dict

Summarize the entire graph.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph
- **name** (*str*) – Name for the graph
- **node_facet_properties** (*Optional[List]*) – A list of properties to facet on. For example, `['provided_by']`
- **edge_facet_properties** (*Optional[List]*) – A list of properties to facet on. For example, `['knowledge_source']`

Returns The stats dictionary

Return type Dict

Remap Node Identifier

The Remap Node Identifier operation can be utilized to remap the `id` field of nodes in a graph.

The main entry point is the `kgx.graph_operations.remap_node_identifier` method that takes an instance of `kgx.graph.base_graph.BaseGraph` along with a `category`, `alternative_property`, and `prefix` as arguments.

kgx.graph_operations.remap_node_identifier

kgx.graph_operations.**remap_node_identifier**(*graph: kgx.graph.base_graph.BaseGraph*, *category: str*, *alternative_property: str*, *prefix=None*) → kgx.graph.base_graph.BaseGraph

Remap a node's 'id' attribute with value from a node's `alternative_property` attribute.

Parameters

- **graph** (`kgx.graph.base_graph.BaseGraph`) – The graph
- **category** (*string*) – category referring to nodes whose 'id' needs to be remapped
- **alternative_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `alternative_property` is a list and the `prefix` indicates which value to pick from the list

Returns The modified graph

Return type *kgx.graph.base_graph.BaseGraph*

Remap Node Property

The Remap Node Property operation can be used to remap the value in a node property with the value from another node property.

kgx.graph_operations.remap_node_property

kgx.graph_operations.**remap_node_property**(*graph: kgx.graph.base_graph.BaseGraph*, *category: str*, *old_property: str*, *new_property: str*) → None

Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **graph** (`kgx.graph.base_graph.BaseGraph`) – The graph
- **category** (*string*) – Category referring to nodes whose property needs to be remapped
- **old_property** (*string*) – old property name whose value needs to be replaced
- **new_property** (*string*) – new property name from which the value is pulled from

Remap Edge Property

The Remap Edge Property operation can be used to remap the value in an edge property with the value from another edge property.

kgx.graph_operations.remap_edge_property

kgx.graph_operations.**remap_edge_property**(*graph: kgx.graph.base_graph.BaseGraph*, *edge_predicate: str*, *old_property: str*, *new_property: str*) → None

Remap the value in an edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph
- **edge_predicate** (*string*) – edge_predicate referring to edges whose property needs to be remapped
- **old_property** (*string*) – Old property name whose value needs to be replaced
- **new_property** (*string*) – New property name from which the value is pulled from

Fold Predicate

The Fold Predicate operation can be use to transform every instance of a predicate to a node property such that if `S -P -> O`,

- `P` becomes the node property name on node `S`
- `O` becomes the value for the node property `P` on node `S`

kgx.graph_operations.fold_predicate

kgx.graph_operations.**fold_predicate**(*graph: kgx.graph.base_graph.BaseGraph*, *predicate: str*, *remove_prefix: bool = False*) → None

Fold predicate as node property where every edge with `predicate` will be folded as a node property.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph
- **predicate** (*str*) – The predicate to fold
- **remove_prefix** (*bool*) – Whether or not to remove prefix from the predicate (`False`, by default)

Unfold Node Property

The Unfold Node Property operation can be used to transform every instance of a node property to a predicate such that if a node `N` has property `P` and value `X`,

- `P` becomes the predicate
- `N` is the subject of the predicate
- `X` becomes the object of the predicate

kgx.graph_operations.unfold_node_property

kgx.graph_operations.**unfold_node_property**(*graph: kgx.graph.base_graph.BaseGraph*, *node_property: str*, *prefix: Optional[str] = None*) → None

Unfold node property as a predicate where every node with `node_property` will be unfolded as an edge.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph
- **node_property** (*str*) – The node property to unfold
- **prefix** (*Optional[str]*) – The prefix to use

Remove Singleton Nodes

The Remove Singleton Nodes operation can be used to remove all nodes in the graph that has a degree of `0`. i.e. they are not participating in any edges.

This operation is typically useful for pruning graphs with isolated nodes before using them in machine learning workflows.

kgx.graph_operations.remove_singleton_nodes

kgx.graph_operations.**remove_singleton_nodes**(*graph: kgx.graph.base_graph.BaseGraph*) → None

Remove singleton nodes (nodes that have a degree of 0) from the graph.

Parameters **graph** (kgx.graph.base_graph.BaseGraph) – The graph

1.2.8 Utilities

The `kgx.utils` module includes all the utility methods used throughout KGX.

KGX Utils

Utility methods that are reused across the codebase.

kgx.utils.kgx_utils

class kgx.utils.kgx_utils.**GraphEntityType**
Bases: enum.Enum

An enumeration.

kgx.utils.kgx_utils.**apply_edge_filters**(*graph: kgx.graph.base_graph.BaseGraph, edge_filters: Dict[str, Union[str, Set]]*) → None
Apply filters to graph and remove edges that do not pass given filters.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph
- **edge_filters** (*Dict[str, Union[str, Set]]*) – Edge filters

kgx.utils.kgx_utils.**apply_filters**(*graph: kgx.graph.base_graph.BaseGraph, node_filters: Dict[str, Union[str, Set]], edge_filters: Dict[str, Union[str, Set]]*) → None
Apply filters to graph and remove nodes and edges that do not pass given filters.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph
- **node_filters** (*Dict[str, Union[str, Set]]*) – Node filters
- **edge_filters** (*Dict[str, Union[str, Set]]*) – Edge filters

kgx.utils.kgx_utils.**apply_graph_operations**(*graph: kgx.graph.base_graph.BaseGraph, operations: List*) → None
Apply graph operations to a given graph.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – An instance of BaseGraph
- **operations** (*List*) – A list of graph operations with configuration

kgx.utils.kgx_utils.**apply_node_filters**(*graph: kgx.graph.base_graph.BaseGraph, node_filters: Dict[str, Union[str, Set]]*) → None
Apply filters to graph and remove nodes that do not pass given filters.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – The graph
- **node_filters** (*Dict[str, Union[str, Set]]*) – Node filters

kgx.utils.kgx_utils.**camelcase_to_sentencecase**(*s: str*) → str
Convert CamelCase to sentence case.

Parameters **s** (*str*) – Input string in CamelCase

Returns string in sentence case form

Return type str

kgx.utils.kgx_utils.**contract**(*uri: str*, *prefix_maps: Optional[List[Dict]] = None*, *fallback: bool = True*) → str

Contract a given URI to a CURIE, based on mappings from *prefix_maps*. If no prefix map is provided then will use defaults from prefixcommons-py.

This method will return the URI as the CURIE if there is no mapping found.

Parameters

- **uri** (*str*) – A URI
- **prefix_maps** (*Optional[List[Dict]]*) – A list of prefix maps to use for mapping
- **fallback** (*bool*) – Determines whether to fallback to default prefix mappings, as determined by *prefixcommons.curie_util*, when URI prefix is not found in *prefix_maps*.

Returns A CURIE corresponding to the URI

Return type str

kgx.utils.kgx_utils.**current_time_in_millis**()

Get current time in milliseconds.

Returns Time in milliseconds

Return type int

kgx.utils.kgx_utils.**expand**(*curie: str*, *prefix_maps: Optional[List[dict]] = None*, *fallback: bool = True*) → str

Expand a given CURIE to an URI, based on mappings from *prefix_map*.

This method will return the CURIE as the IRI if there is no mapping found.

Parameters

- **curie** (*str*) – A CURIE
- **prefix_maps** (*Optional[List[dict]]*) – A list of prefix maps to use for mapping
- **fallback** (*bool*) – Determines whether to fallback to default prefix mappings, as determined by *prefixcommons.curie_util*, when CURIE prefix is not found in *prefix_maps*.

Returns A URI corresponding to the CURIE

Return type str

kgx.utils.kgx_utils.**format_biolink_category**(*s: str*) → str

Convert a sentence case Biolink category name to a proper Biolink CURIE with the category itself in CamelCase form.

Parameters **s** (*str*) – Input string in sentence case

Returns a proper Biolink CURIE

Return type str

kgx.utils.kgx_utils.**generate_edge_identifiers**(*graph: kgx.graph.base_graph.BaseGraph*)

Generate unique identifiers for edges in a graph that do not have an `id` field.

Parameters **graph** (kgx.graph.base_graph.BaseGraph) –

kgx.utils.kgx_utils.**generate_edge_key**(*s: str*, *edge_predicate: str*, *o: str*) → str

Generates an edge key based on a given subject, predicate, and object.

Parameters

- **s** (*str*) – Subject

- **edge_predicate** (*str*) – Edge label
- **o** (*str*) – Object

Returns Edge key as a string

Return type str

kgx.utils.kgx_utils.**generate_uuid**()
Generates a UUID.

Returns A UUID

Return type str

kgx.utils.kgx_utils.**get_biolink_ancestors**(*name: str*)
Get ancestors for a given Biolink class.

Parameters **name** (*str*) –

Returns A list of ancestors

Return type List

kgx.utils.kgx_utils.**get_biolink_element**(*name*) → Optional[linkml_runtime.linkml_model.meta.Element]
Get Biolink element for a given name, where name can be a class, slot, or relation.

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

Returns An instance of linkml_model.meta.Element

Return type Optional[linkml_model.meta.Element]

kgx.utils.kgx_utils.**get_biolink_property_types**() → Dict
Get all Biolink property types. This includes both node and edges properties.

Returns A dict containing all Biolink property and their types

Return type Dict

kgx.utils.kgx_utils.**get_cache**(*maxsize=10000*)
Get an instance of cachetools.cache

Parameters **maxsize** (*int*) – The max size for the cache (10000, by default)

Returns An instance of cachetools.cache

Return type cachetools.cache

kgx.utils.kgx_utils.**get_curie_lookup_service**()
Get an instance of kgx.curie_lookup_service.CurieLookupService

Returns An instance of CurieLookupService

Return type *kgx.curie_lookup_service.CurieLookupService*

kgx.utils.kgx_utils.**get_prefix_prioritization_map**() → Dict[str, List]
Get prefix prioritization map as defined in Biolink Model.

Returns

Return type Dict[str, List]

kgx.utils.kgx_utils.**get_toolkit**(*biolink_release: Optional[str] = None*) → bmt.toolkit.Toolkit
Get an instance of bmt.Toolkit If there no instance defined, then one is instantiated and returned.

Parameters **biolink_release** (*Optional[str]*) – URL to (Biolink) Model Schema to be used for validated (default: None, use default Biolink Model Toolkit schema)

kgx.utils.kgx_utils.**get_type_for_property**(*p: str*) → str
Get type for a property.

TODO: Move this to biolink-model-default_toolkit

Parameters **p** (*str*) –

Returns The type for a given property

Return type str

kgx.utils.kgx_utils.**is_null**(*item: Any*) → bool
Checks if a given item is null or correspond to null.

This method checks for: `None`, `numpy.nan`, `pandas.NA`, `pandas.NaT`, and ``

Parameters **item** (*Any*) – The item to check

Returns Whether the given item is null or not

Return type bool

kgx.utils.kgx_utils.**prepare_data_dict**(*d1: Dict*, *d2: Dict*, *preserve: bool = True*) → Dict
Given two dict objects, make a new dict object that is the intersection of the two.

If a key is known to be multivalued then it's value is converted to a list. If a key is already multivalued then it is updated with new values. If a key is single valued, and a new unique value is found then the existing value is converted to a list and the new value is appended to this list.

Parameters

- **d1** (*Dict*) – Dict object
- **d2** (*Dict*) – Dict object
- **preserve** (*bool*) – Whether or not to preserve values for conflicting keys

Returns The intersection of d1 and d2

Return type Dict

kgx.utils.kgx_utils.**remove_null**(*input: Any*) → Any
Remove any null values from input.

Parameters **input** (*Any*) – Can be a str, list or dict

Returns The input without any null values

Return type Any

kgx.utils.kgx_utils.**sanitize_import**(*data: Dict*) → Dict
Sanitize key-value pairs in dictionary.

Parameters **data** (*Dict*) – A dictionary containing key-value pairs

Returns A dictionary containing processed key-value pairs

Return type Dict

kgx.utils.kgx_utils.**sentencecase_to_camelcase**(*s: str*) → str
Convert sentence case to CamelCase.

Parameters **s** (*str*) – Input string in sentence case

Returns string in CamelCase form

Return type str

kgx.utils.kgx_utils.**sentencecase_to_snakecase**(*s: str*) → str
Convert sentence case to snake_case.

Parameters **s** (*str*) – Input string in sentence case

Returns string in snake_case form

Return type str

kgx.utils.kgx_utils.**snakecase_to_sentencecase**(*s: str*) → str
Convert snake_case to sentence case.

Parameters **s** (*str*) – Input string in snake_case

Returns string in sentence case form

Return type str

kgx.utils.kgx_utils.**validate_edge**(*edge: Dict*) → Dict
Given an edge as a dictionary, check for required properties. This method will return the edge dictionary with default assumptions applied, if any.

Parameters **edge** (*Dict*) – An edge represented as a dict

Returns An edge represented as a dict, with default assumptions applied.

Return type Dict

kgx.utils.kgx_utils.**validate_node**(*node: Dict*) → Dict
Given a node as a dictionary, check for required properties. This method will return the node dictionary with default assumptions applied, if any.

Parameters **node** (*Dict*) – A node represented as a dict

Returns A node represented as a dict, with default assumptions applied.

Return type Dict

Graph Utils

Utility methods for working with graphs.

kgx.utils.graph_utils

kgx.utils.graph_utils.**curie_lookup**(*curie: str*) → Optional[str]
Given a CURIE, find its label.

This method first does a lookup in predefined maps. If none found, it makes use of CurieLookupService to look for the CURIE in a set of preloaded ontologies.

Parameters **curie** (*str*) – A CURIE

Returns The label corresponding to the given CURIE

Return type Optional[str]

kgx.utils.graph_utils.**get_ancestors**(*graph: kgx.graph.base_graph.BaseGraph*, *node: str*, *relations: List[str] = None*) → List[str]
Return all *ancestors* of specified node, filtered by `relations`.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – Graph to traverse

- **node** (*str*) – node identifier
- **relations** (*List[str]*) – list of relations

Returns A list of ancestor nodes

Return type List[str]

kgx.utils.graph_utils.**get_category_via_superclass**(*graph: kgx.graph.base_graph.BaseGraph*, *curie: str*, *load_ontology: bool = True*) → Set[str]

Get category for a given CURIE by tracing its superclass, via `subclass_of` hierarchy, and getting the most appropriate category based on the superclass.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – Graph to traverse
- **curie** (*str*) – Input CURIE
- **load_ontology** (*bool*) – Determines whether to load ontology, based on CURIE prefix, or to simply rely on `subclass_of` hierarchy from graph

Returns A set containing one (or more) category for the given CURIE

Return type Set[str]

kgx.utils.graph_utils.**get_parents**(*graph: kgx.graph.base_graph.BaseGraph*, *node: str*, *relations: List[str] = None*) → List[str]

Return all direct *parents* of a specified node, filtered by `relations`.

Parameters

- **graph** (kgx.graph.base_graph.BaseGraph) – Graph to traverse
- **node** (*str*) – node identifier
- **relations** (*List[str]*) – list of relations

Returns A list of parent node(s)

Return type List[str]

RDF Utils

Utility methods that are used for handling RDF.

kgx.utils.rdf_utils

kgx.utils.rdf_utils.**get_biolink_element**(*prefix_manager: kgx.prefix_manager.PrefixManager*, *predicate: Any*) → Optional[linkml_runtime.linkml_model.meta.Element]

Returns a Biolink Model element for a given predicate.

Parameters

- **prefix_manager** (PrefixManager) – An instance of prefix manager
- **predicate** (*Any*) – The CURIE of a predicate

Returns The corresponding Biolink Model element

Return type Optional[Element]

kgx.utils.rdf_utils.**infer_category**(*iri: rdflib.term.URIRef*, *rdfgraph: rdflib.graph.Graph*) → Optional[List]

Infer category for a given iri by traversing rdfgraph.

Parameters

- **iri** (*rdflib.term.URIRef*) – IRI
- **rdfgraph** (*rdflib.Graph*) – A graph to traverse

Returns A list of category corresponding to the given IRI

Return type Optional[List]

kgx.utils.rdf_utils.**process_predicate**(*prefix_manager: kgx.prefix_manager.PrefixManager*, *p: Union[rdflib.term.URIRef, str]*, *predicate_mapping: Optional[Dict] = None*) → Tuple

Process a predicate where the method checks if there is a mapping in Biolink Model.

Parameters

- **prefix_manager** (PrefixManager) – An instance of prefix manager
- **p** (*Union[URIRef, str]*) – The predicate
- **predicate_mapping** (*Optional[Dict]*) – Predicate mappings

Returns A tuple that contains the Biolink CURIE (if available), the Biolink slot_uri CURIE (if available), the CURIE form of p, the reference of p

Return type Tuple[str, str, str, str]

1.2.9 Prefix Manager

In KGX, the `PrefixManager` acts as a central resource for,

- CURIE to IRI expansion
- IRI to CURIE contraction

Under the hood, `PrefixManager` makes use of prefixcommons-py.

Each time the `PrefixManager` class is initialized, it makes use of the Biolink Model JSON-LD context for a default set of prefix to IRI mappings.

These defaults can be overridden by using `update_prefix_map` and providing your custom mappings.

kgx.prefix_manager

class kgx.prefix_manager.**PrefixManager**(*url: str = None*)

Bases: `object`

Manages prefix mappings.

These include mappings for CURIEs such as GO:0008150, as well as shortforms such as biolink types such as Disease

__init__(*url: str = None*)

Initialize an instance of PrefixManager.

Parameters **url** (*str*) – The URL from which to read a JSON-LD context for prefix mappings

contract(*uri: str*, *fallback: bool = True*) → Optional[str]

Contract a given URI to a CURIE, based on mappings from *prefix_map*.

Parameters

- **uri** (*str*) – A URI
- **fallback** (*bool*) – Determines whether to fallback to default prefix mappings, as determined by *prefixcommons.curie_util*, when URI prefix is not found in *reverse_prefix_map*.

Returns A CURIE corresponding to the URI

Return type Optional[str]

expand(*curie: str*, *fallback: bool = True*) → str
Expand a given CURIE to an URI, based on mappings from *prefix_map*.

Parameters

- **curie** (*str*) – A CURIE
- **fallback** (*bool*) – Determines whether to fallback to default prefix mappings, as determined by *prefixcommons.curie_util*, when CURIE prefix is not found in *prefix_map*.

Returns A URI corresponding to the CURIE

Return type str

static get_prefix(*curie: str*) → Optional[str]
Get the prefix from a given CURIE.

Parameters **curie** (*str*) – The CURIE

Returns The CURIE prefix

Return type str

static get_reference(*curie: str*) → Optional[str]
Get the reference of a given CURIE.

Parameters **curie** (*str*) – The CURIE

Returns The reference of a CURIE

Return type Optional[str]

static is_curie(*s: str*) → bool
Check if a given string is a CURIE.

Parameters **s** (*str*) – A string

Returns Whether or not the given string is a CURIE

Return type bool

static is_iri(*s: str*) → bool
Check if a given string as an IRI.

Parameters **s** (*str*) – A string

Returns Whether or not the given string is an IRI.

Return type bool

set_prefix_map(*m: Dict*) → None
Populate *prefix_map* with contents from a JSON-LD context from self.url

Parameters **m** (*dict*) – Dictionary of prefix to URI mappings

update_prefix_map(*m: Dict[str, str]*) → None
Update prefix maps with new mappings.

Parameters **m** (*Dict*) – New prefix to IRI mappings

update_reverse_prefix_map(*m: Dict[str, str]*) → None
Update reverse prefix maps with new mappings.

Parameters **m** (*Dict*) – New IRI to prefix mappings

1.2.10 CURIE Lookup Service

The CURIE Lookup Service supports the ability to lookup labels for a given CURIE.

It does so by pre-loading all the relevant ontologies when the `CurieLookupService` class is initialized, where only the terms and their `rdfs:label` are loaded into a separate graph specifically for the purpose of lookup.

The required ontologies are defined in the KGX `config.yml`.

kgx.curie_lookup_service

class kgx.curie_lookup_service.**CurieLookupService**(*curie_map: dict = None*)
Bases: `object`

A service to lookup label for a given CURIE.

load_ontologies()
Load all required ontologies.

1.3 KGX Format

The KGX format is a serialization of Biolink Model compliant knowledge graphs.

The complete up-to-date format specification can be found here.

1.4 Examples

There are two modes for using KGX:

- Using KGX as a module
- Using KGX CLI

1.4.1 Using KGX as a module

KGX provides a variety of functionality that can be achieved by your script by importing KGX as a module and calling the relevant classes/methods as needed.

Examples on how to use the KGX as a module can be found in examples folder. Some additional insights on usage may also be obtained by examination of the tests folder.

1.4.2 Using KGX CLI

The KGX CLI is a way of accessing KGX's functionality directly from the command line.

Currently, the CLI supports the following operations:

graph-summary

Summarizes a graph and generate a YAML report regarding the composition of node and edge types in the graph.

```
kgx graph-summary --input-format tsv \
                  --output graph_stats.yaml \
                  --report-type kgx-map \
                  --error-log graph_stats.err \
                  tests/resources/graph_nodes.tsv tests/resources/graph_edges.tsv
```

An alternate summary of a graph generates a TRAPI 1.*-compliant meta knowledge graph ("content metadata") JSON report:

```
kgx graph-summary --input-format tsv \
                  --output content_metadata.json \
                  --report-type meta-knowledge-graph \
                  --error-log graph_stats.err \
                  tests/resources/graph_nodes.tsv tests/resources/graph_edges.tsv
```

Some basic validation is done during **graph-summary** operation, with detected errors reported on the `--error_log` (default: `stderr`). For more complete graph validation, the **validate** command (below) may be used.

validate

Validate a graph for Biolink Model compliance and generate a report for nodes and edges that are not compliant (if any).

```
kgx validate --input-format tsv \
             tests/resources/test_nodes.tsv tests/resources/test_edges.tsv
```

neo4j-download

Download a (sub)graph from a local or remote Neo4j instance.

```
kgx neo4j-download --uri http://localhost:7474 \
                   --username neo4j \
                   --password admin \
                   --output neo_graph_download \
                   --output-format tsv
```

neo4j-upload

Upload a (sub)graph to a clean local or remote Neo4j instance.

Note: This operation expects the Neo4j instance to be empty. This operation does not support updating an existing Neo4j graph. Writing to an existing graph may lead to side effects.

```
kgx neo4j-upload --uri http://localhost:7474 \
                 --username neo4j \
                 --password admin \
                 --input-format tsv \
                 tests/resources/test_nodes.tsv tests/resources/test_edges.tsv
```

transform

Transform a graph from one serialization to another (including a knowledge source InfoRes rewrite).

```
kgx transform --input-format tsv \
              --output test_graph.json \
              --output-format json \
              --knowledge-sources aggregator_knowledge_source "string,string␣
↪database" \
              --knowledge-sources aggregator_knowledge_source "go,gene ontology" \
              tests/resources/graph_nodes.tsv tests/resources/graph_edges.tsv
```

Alternatively, you can also perform transformation driven by a YAML.

A sample of the merge configuration can be found here.

```
kgx transform --transform-config transform.yaml
```

merge

Merge two (or more) graphs as defined by a YAML merge configuration.

A sample of the merge configuration can be found here

```
kgx merge --merge-config merge.yaml
```

PYTHON MODULE INDEX

k

kgx.cli.cli_utils, 9
kgx.curie_lookup_service, 99
kgx.graph.base_graph, 15
kgx.graph.nx_graph, 20
kgx.graph_operations.clique_merge, 75
kgx.graph_operations.graph_merge, 81
kgx.graph_operations.summarize_graph, 83
kgx.prefix_manager, 97
kgx.sink.graph_sink, 63
kgx.sink.json_sink, 65
kgx.sink.jsonl_sink, 65
kgx.sink.neo_sink, 67
kgx.sink.rdf_sink, 68
kgx.sink.sink, 63
kgx.sink.trapi_sink, 66
kgx.sink.tsv_sink, 64
kgx.source.graph_source, 30
kgx.source.json_source, 34
kgx.source.jsonl_source, 37
kgx.source.neo_source, 47
kgx.source.obograph_source, 42
kgx.source.owl_source, 55
kgx.source.rdf_source, 51
kgx.source.source, 29
kgx.source.sparql_source, 58
kgx.source.sssom_source, 44
kgx.source.trapi_source, 39
kgx.source.tsv_source, 32
kgx.transformer, 27
kgx.utils.graph_utils, 95
kgx.utils.kgx_utils, 91
kgx.utils.rdf_utils, 96
kgx.validator, 70

INDEX

Symbols

__call__() (*kgx.graph_operations.summarize_graph.GraphSummary method*), 84
__call__() (*kgx.validator.Validator method*), 71
__init__() (*kgx.graph_operations.summarize_graph.GraphSummary method*), 84
__init__() (*kgx.graph_operations.summarize_graph.GraphSummary.Category method*), 83
__init__() (*kgx.prefix_manager.PrefixManager method*), 97
--biolink-release <biolink_release>
    kgx-validate command line option, 9
--destination <destination>
    kgx-merge command line option, 6
--edge-facet-properties <edge_facet_properties>
    kgx-graph-summary command line option, 5
--edge-filters <edge_filters>
    kgx-neo4j-download command line option, 7
    kgx-neo4j-upload command line option, 7
    kgx-transform command line option, 8
--error-log <error_log>
    kgx-graph-summary command line option, 5
--graph-name <graph_name>
    kgx-graph-summary command line option, 5
--infores-catalog <infores_catalog>
    kgx-transform command line option, 8
--input-compression <input_compression>
    kgx-graph-summary command line option, 5
    kgx-neo4j-upload command line option, 7
    kgx-transform command line option, 8
    kgx-validate command line option, 9
--input-format <input_format>
    kgx-graph-summary command line option, 5
    kgx-neo4j-upload command line option, 7
    kgx-transform command line option, 8
    kgx-validate command line option, 9
--knowledge-sources <knowledge_sources>
    kgx-transform command line option, 8
--merge-config <merge_config>
    kgx-merge command line option, 6
--node-facet-properties <node_facet_properties>
    kgx-graph-summary command line option, 5
--node-filters <node_filters>
    kgx-neo4j-download command line option, 7
    kgx-neo4j-upload command line option, 7
    kgx-transform command line option, 8
--output <output>
    kgx-graph-summary command line option, 5
    kgx-neo4j-download command line option, 6
    kgx-transform command line option, 8
    kgx-validate command line option, 9
--output-compression <output_compression>
    kgx-neo4j-download command line option, 6
    kgx-transform command line option, 8
--output-format <output_format>
    kgx-neo4j-download command line option, 6
    kgx-transform command line option, 8
--password <password>
    kgx-neo4j-download command line option, 6
    kgx-neo4j-upload command line option, 7
--processes <processes>

kgx-merge command line option, 6
kgx-transform command line option, 8
--report-format <report_format>
kgx-graph-summary command line option, 5
--report-type <report_type>
kgx-graph-summary command line option, 5
--source <source>
kgx-merge command line option, 6
kgx-transform command line option, 8
--stream
kgx-graph-summary command line option, 5
kgx-neo4j-download command line option, 6
kgx-neo4j-upload command line option, 7
kgx-transform command line option, 8
kgx-validate command line option, 9
--transform-config <transform_config>
kgx-transform command line option, 8
--uri <uri>
kgx-neo4j-download command line option, 6
kgx-neo4j-upload command line option, 7
--username <username>
kgx-neo4j-download command line option, 6
kgx-neo4j-upload command line option, 7
--version
kgx command line option, 5
-b
kgx-validate command line option, 9
-c
kgx-graph-summary command line option, 5
kgx-neo4j-upload command line option, 7
kgx-transform command line option, 8
kgx-validate command line option, 9
-d
kgx-neo4j-download command line option, 6
kgx-transform command line option, 8
-e
kgx-neo4j-download command line option, 7
kgx-neo4j-upload command line option, 7
kgx-transform command line option, 8
-f
kgx-graph-summary command line option, 5
kgx-neo4j-download command line option, 6
kgx-transform command line option, 8
-i
kgx-graph-summary command line option, 5
kgx-neo4j-upload command line option, 7
kgx-transform command line option, 8
kgx-validate command line option, 9
-k
kgx-transform command line option, 8
-l
kgx-graph-summary command line option, 5
kgx-neo4j-download command line option, 6
kgx-neo4j-upload command line option, 7
-n
kgx-graph-summary command line option, 5
kgx-neo4j-download command line option, 7
kgx-neo4j-upload command line option, 7
kgx-transform command line option, 8
-o
kgx-graph-summary command line option, 5
kgx-neo4j-download command line option, 6
kgx-transform command line option, 8
kgx-validate command line option, 9
-p
kgx-merge command line option, 6
kgx-neo4j-download command line option, 6
kgx-neo4j-upload command line option, 7
kgx-transform command line option, 8
-r
kgx-graph-summary command line option, 5
-s
kgx-graph-summary command line option, 5
kgx-neo4j-download command line option, 6
kgx-neo4j-upload command line option, 7
kgx-validate command line option, 9

-u
    kgx-neo4j-download command line option, 6
    kgx-neo4j-upload command line option, 7

A

add_all_edges() (*in module kgx.graph_operations.graph_merge*), 81
add_all_nodes() (*in module kgx.graph_operations.graph_merge*), 81
add_edge() (*kgx.graph.base_graph.BaseGraph method*), 15
add_edge() (*kgx.graph.nx_graph.NxGraph method*), 20
add_edge() (*kgx.source.owl_source.OwlSource method*), 55
add_edge() (*kgx.source.rdf_source.RdfSource method*), 51
add_edge() (*kgx.source.sparql_source.SparqlSource method*), 58
add_edge_attribute() (*kgx.graph.base_graph.BaseGraph method*), 15
add_edge_attribute() (*kgx.graph.nx_graph.NxGraph method*), 20
add_node() (*kgx.graph.base_graph.BaseGraph method*), 16
add_node() (*kgx.graph.nx_graph.NxGraph method*), 20
add_node() (*kgx.source.owl_source.OwlSource method*), 55
add_node() (*kgx.source.rdf_source.RdfSource method*), 51
add_node() (*kgx.source.sparql_source.SparqlSource method*), 59
add_node_attribute() (*kgx.graph.base_graph.BaseGraph method*), 16
add_node_attribute() (*kgx.graph.nx_graph.NxGraph method*), 20
add_node_attribute() (*kgx.source.owl_source.OwlSource method*), 55
add_node_attribute() (*kgx.source.rdf_source.RdfSource method*), 51
add_node_attribute() (*kgx.source.sparql_source.SparqlSource method*), 59
add_node_stat() (*kgx.graph_operations.summarize_graph.GraphSummary method*), 85
analyse_edge() (*kgx.graph_operations.summarize_graph.GraphSumm method*), 85
analyse_node() (*kgx.graph_operations.summarize_graph.GraphSumm method*), 85
analyse_node_category() (*kgx.graph_operations.summarize_graph.GraphSummary.Catego method*), 84
apply_edge_filters() (*in module kgx.utils.kgx_utils*), 91
apply_filters() (*in module kgx.utils.kgx_utils*), 91
apply_graph_operations() (*in module kgx.utils.kgx_utils*), 91
apply_node_filters() (*in module kgx.utils.kgx_utils*), 91
apply_operations() (*in module kgx.cli.cli_utils*), 9

B

BaseGraph (*class in kgx.graph.base_graph*), 15
build_cliques() (*in module kgx.graph_operations.clique_merge*), 75

C

camelcase_to_sentencecase() (*in module kgx.utils.kgx_utils*), 91
check_all_categories() (*in module kgx.graph_operations.clique_merge*), 75
check_categories() (*in module kgx.graph_operations.clique_merge*), 76
check_edge_filter() (*kgx.source.graph_source.GraphSource method*), 30
check_edge_filter() (*kgx.source.json_source.JsonSource method*), 34
check_edge_filter() (*kgx.source.jsonl_source.JsonlSource method*), 37
check_edge_filter() (*kgx.source.neo_source.NeoSource method*), 47
check_edge_filter() (*kgx.source.obograph_source.ObographSource method*), 42
check_edge_filter() (*kgx.source.owl_source.OwlSource method*), 55
check_edge_filter() (*kgx.source.rdf_source.RdfSource method*), 52
check_edge_filter() (*kgx.source.source.Source method*), 29
check_edge_filter() (*kgx.source.sparql_source.SparqlSource method*), 59

check_edge_filter() (*kgx.source.sssom_source.SssomSource method*), 44
check_edge_filter() (*kgx.source.trapi_source.TrapiSource method*), 39
check_edge_filter() (*kgx.source.tsv_source.TsvSource method*), 32
check_node_filter() (*kgx.source.graph_source.GraphSource method*), 30
check_node_filter() (*kgx.source.json_source.JsonSource method*), 34
check_node_filter() (*kgx.source.jsonl_source.JsonlSource method*), 37
check_node_filter() (*kgx.source.neo_source.NeoSource method*), 47
check_node_filter() (*kgx.source.obograph_source.ObographSource method*), 42
check_node_filter() (*kgx.source.owl_source.OwlSource method*), 56
check_node_filter() (*kgx.source.rdf_source.RdfSource method*), 52
check_node_filter() (*kgx.source.source.Source method*), 29
check_node_filter() (*kgx.source.sparql_source.SparqlSource method*), 59
check_node_filter() (*kgx.source.sssom_source.SssomSource method*), 45
check_node_filter() (*kgx.source.trapi_source.TrapiSource method*), 39
check_node_filter() (*kgx.source.tsv_source.TsvSource method*), 32
clear() (*kgx.graph.base_graph.BaseGraph method*), 16
clear() (*kgx.graph.nx_graph.NxGraph method*), 21
clear_graph_metadata() (*kgx.source.graph_source.GraphSource method*), 30
clear_graph_metadata() (*kgx.source.json_source.JsonSource method*), 35
clear_graph_metadata() (*kgx.source.jsonl_source.JsonlSource method*), 37
clear_graph_metadata() (*kgx.source.neo_source.NeoSource method*), 47
clear_graph_metadata() (*kgx.source.obograph_source.ObographSource method*), 42
clear_graph_metadata() (*kgx.source.owl_source.OwlSource method*), 56
clear_graph_metadata() (*kgx.source.rdf_source.RdfSource method*), 52
clear_graph_metadata() (*kgx.source.source.Source method*), 29
clear_graph_metadata() (*kgx.source.sparql_source.SparqlSource method*), 59
clear_graph_metadata() (*kgx.source.sssom_source.SssomSource method*), 45
clear_graph_metadata() (*kgx.source.trapi_source.TrapiSource method*), 39
clear_graph_metadata() (*kgx.source.tsv_source.TsvSource method*), 32
clique_merge() (*in module kgx.graph_operations.clique_merge*), 76
consolidate_edges() (*in module kgx.graph_operations.clique_merge*), 76
contract() (*in module kgx.utils.kgx_utils*), 91
contract() (*kgx.prefix_manager.PrefixManager method*), 97
count() (*kgx.source.neo_source.NeoSource method*), 47
create_constraint_query() (*kgx.sink.neo_sink.NeoSink static method*), 67
create_constraints() (*kgx.sink.neo_sink.NeoSink method*), 67
curie_lookup() (*in module kgx.utils.graph_utils*), 95
CurieLookupService (*class in kgx.curie_lookup_service*), 99
current_time_in_millis() (*in module kgx.utils.kgx_utils*), 92

D

degree() (*kgx.graph.base_graph.BaseGraph method*), 16
degree() (*kgx.graph.nx_graph.NxGraph method*), 21

dereify() (*kgx.source.owl_source.OwlSource method*), 56
dereify() (*kgx.source.rdf_source.RdfSource method*), 52
dereify() (*kgx.source.sparql_source.SparqlSource method*), 59

E

edges() (*kgx.graph.base_graph.BaseGraph method*), 16
edges() (*kgx.graph.nx_graph.NxGraph method*), 21
edges_iter() (*kgx.graph.base_graph.BaseGraph method*), 16
edges_iter() (*kgx.graph.nx_graph.NxGraph method*), 21
elect_leader() (*in module kgx.graph_operations.clique_merge*), 77
ErrorType (*class in kgx.validator*), 70
expand() (*in module kgx.utils.kgx_utils*), 92
expand() (*kgx.prefix_manager.PrefixManager method*), 98

F

finalize() (*kgx.sink.graph_sink.GraphSink method*), 64
finalize() (*kgx.sink.json_sink.JsonSink method*), 65
finalize() (*kgx.sink.jsonl_sink.JsonlSink method*), 66
finalize() (*kgx.sink.neo_sink.NeoSink method*), 67
finalize() (*kgx.sink.rdf_sink.RdfSink method*), 68
finalize() (*kgx.sink.sink.Sink method*), 63
finalize() (*kgx.sink.trapi_sink.TrapiSink method*), 66
finalize() (*kgx.sink.tsv_sink.TsvSink method*), 64
fold_predicate() (*in module kgx.graph_operations*), 89
format_biolink_category() (*in module kgx.utils.kgx_utils*), 92
format_edge_filter() (*kgx.source.neo_source.NeoSource static method*), 47
format_node_filter() (*kgx.source.neo_source.NeoSource static method*), 48

G

generate_edge_identifiers() (*in module kgx.utils.kgx_utils*), 92
generate_edge_key() (*in module kgx.utils.kgx_utils*), 92
generate_graph_stats() (*in module kgx.graph_operations.summarize_graph*), 87
generate_unwind_edge_query() (*kgx.sink.neo_sink.NeoSink static method*), 67
generate_unwind_node_query() (*kgx.sink.neo_sink.NeoSink static method*), 67
generate_uuid() (*in module kgx.utils.kgx_utils*), 93
get_all_prefixes() (*kgx.validator.Validator static method*), 71
get_ancestors() (*in module kgx.utils.graph_utils*), 95
get_biolink_ancestors() (*in module kgx.utils.kgx_utils*), 93
get_biolink_element() (*in module kgx.utils.kgx_utils*), 93
get_biolink_element() (*in module kgx.utils.rdf_utils*), 96
get_biolink_element() (*kgx.sink.rdf_sink.RdfSink method*), 68
get_biolink_element() (*kgx.source.owl_source.OwlSource method*), 56
get_biolink_element() (*kgx.source.rdf_source.RdfSource method*), 52
get_biolink_element() (*kgx.source.sparql_source.SparqlSource method*), 60
get_biolink_property_types() (*in module kgx.utils.kgx_utils*), 93
get_cache() (*in module kgx.utils.kgx_utils*), 93
get_category() (*kgx.graph_operations.summarize_graph.GraphSumm method*), 85
get_category() (*kgx.source.obograph_source.ObographSource method*), 42
get_category_from_equivalence() (*in module kgx.graph_operations.clique_merge*), 77
get_category_via_superclass() (*in module kgx.utils.graph_utils*), 96
get_cid() (*kgx.graph_operations.summarize_graph.GraphSummary.Ca method*), 84
get_clique_category() (*in module kgx.graph_operations.clique_merge*), 78
get_count() (*kgx.graph_operations.summarize_graph.GraphSummary. method*), 84
get_count_by_id_prefixes() (*kgx.graph_operations.summarize_graph.GraphSummary.Catego method*), 84
get_curie_lookup_service() (*in module kgx.utils.kgx_utils*), 93
get_edge() (*kgx.graph.base_graph.BaseGraph method*), 16
get_edge() (*kgx.graph.nx_graph.NxGraph method*), 21

get_edge_attributes() (kgx.graph.base_graph.BaseGraph static method), 16
get_edge_attributes() (kgx.graph.nx_graph.NxGraph static method), 21
get_edges() (kgx.source.neo_source.NeoSource method), 48
get_error_messages() (kgx.validator.Validator method), 71
get_facet_counts() (kgx.graph_operations.summarize_graph.GraphSummary method), 85
get_graph_summary() (kgx.graph_operations.summarize_graph.GraphSummary method), 86
get_id_prefixes() (kgx.graph_operations.summarize_graph.GraphSummary.Category method), 84
get_infores_catalog() (kgx.source.graph_source.GraphSource method), 30
get_infores_catalog() (kgx.source.json_source.JsonSource method), 35
get_infores_catalog() (kgx.source.jsonl_source.JsonlSource method), 37
get_infores_catalog() (kgx.source.neo_source.NeoSource method), 48
get_infores_catalog() (kgx.source.obograph_source.ObographSource method), 42
get_infores_catalog() (kgx.source.owl_source.OwlSource method), 56
get_infores_catalog() (kgx.source.rdf_source.RdfSource method), 52
get_infores_catalog() (kgx.source.source.Source method), 29
get_infores_catalog() (kgx.source.sparql_source.SparqlSource method), 60
get_infores_catalog() (kgx.source.sssom_source.SssomSource method), 45
get_infores_catalog() (kgx.source.trapi_source.TrapiSource method), 39
get_infores_catalog() (kgx.source.tsv_source.TsvSource method), 32
get_infores_catalog() (kgx.transformer.Transformer method), 27
get_input_file_types() (in module kgx.cli.cli_utils), 9
get_leader_by_annotation() (in module kgx.graph_operations.clique_merge), 78
get_leader_by_prefix_priority() (in module kgx.graph_operations.clique_merge), 78
get_leader_by_sort() (in module kgx.graph_operations.clique_merge), 79
get_name() (kgx.graph_operations.summarize_graph.GraphSummary method), 86
get_name() (kgx.graph_operations.summarize_graph.GraphSummary.C method), 84
get_node() (kgx.graph.base_graph.BaseGraph method), 17
get_node() (kgx.graph.nx_graph.NxGraph method), 21
get_node_attributes() (kgx.graph.base_graph.BaseGraph static method), 17
get_node_attributes() (kgx.graph.nx_graph.NxGraph static method), 21
get_node_stats() (kgx.graph_operations.summarize_graph.GraphSu method), 86
get_nodes() (kgx.source.neo_source.NeoSource method), 48
get_output_file_types() (in module kgx.cli.cli_utils), 9
get_pages() (kgx.source.neo_source.NeoSource method), 48
get_parents() (in module kgx.utils.graph_utils), 96
get_prefix() (kgx.prefix_manager.PrefixManager static method), 98
get_prefix_prioritization_map() (in module kgx.utils.kgx_utils), 93
get_reference() (kgx.prefix_manager.PrefixManager static method), 98
get_report_format_types() (in module kgx.cli.cli_utils), 9
get_required_edge_properties() (kgx.validator.Validator static method), 71
get_required_node_properties() (kgx.validator.Validator static method), 71
get_sink() (kgx.transformer.Transformer method), 27
get_source() (kgx.transformer.Transformer method), 27
get_toolkit() (in module kgx.utils.kgx_utils), 93
get_type_for_property() (in module kgx.utils.kgx_utils), 93
graph_summary() (in module kgx.cli.cli_utils), 10
GraphEntityType (class in kgx.utils.kgx_utils), 91

GraphSink (*class in kgx.sink.graph_sink*), 63
GraphSource (*class in kgx.source.graph_source*), 30
GraphSummary (*class in kgx.graph_operations.summarize_graph*), 83
GraphSummary.Category (*class in kgx.graph_operations.summarize_graph*), 83
gs_default() (*in module kgx.graph_operations.summarize_graph*), 87

H

has_edge() (*kgx.graph.base_graph.BaseGraph method*), 17
has_edge() (*kgx.graph.nx_graph.NxGraph method*), 22
has_node() (*kgx.graph.base_graph.BaseGraph method*), 17
has_node() (*kgx.graph.nx_graph.NxGraph method*), 22

I

in_edges() (*kgx.graph.base_graph.BaseGraph method*), 17
in_edges() (*kgx.graph.nx_graph.NxGraph method*), 22
infer_category() (*in module kgx.utils.rdf_utils*), 96
INPUTS
    kgx-graph-summary command line option, 6
    kgx-neo4j-upload command line option, 7
    kgx-transform command line option, 8
    kgx-validate command line option, 9
is_curie() (*kgx.prefix_manager.PrefixManager static method*), 98
is_iri() (*kgx.prefix_manager.PrefixManager static method*), 98
is_null() (*in module kgx.utils.kgx_utils*), 94

J

json_object() (*kgx.graph_operations.summarize_graph.GraphSummary.Category method*), 84
JsonlSink (*class in kgx.sink.jsonl_sink*), 65
JsonlSource (*class in kgx.source.jsonl_source*), 37
JsonSink (*class in kgx.sink.json_sink*), 65
JsonSource (*class in kgx.source.json_source*), 34

K

kgx command line option
    --version, 5
kgx.cli.cli_utils (*module*), 9
kgx.curie_lookup_service (*module*), 99
kgx.graph.base_graph (*module*), 15
kgx.graph.nx_graph (*module*), 20
kgx.graph_operations.clique_merge (*module*), 75
kgx.graph_operations.graph_merge (*module*), 81
kgx.graph_operations.summarize_graph (*module*), 83
kgx.prefix_manager (*module*), 97
kgx.sink.graph_sink (*module*), 63
kgx.sink.json_sink (*module*), 65
kgx.sink.jsonl_sink (*module*), 65
kgx.sink.neo_sink (*module*), 67
kgx.sink.rdf_sink (*module*), 68
kgx.sink.sink (*module*), 63
kgx.sink.trapi_sink (*module*), 66
kgx.sink.tsv_sink (*module*), 64
kgx.source.graph_source (*module*), 30
kgx.source.json_source (*module*), 34
kgx.source.jsonl_source (*module*), 37
kgx.source.neo_source (*module*), 47
kgx.source.obograph_source (*module*), 42
kgx.source.owl_source (*module*), 55
kgx.source.rdf_source (*module*), 51
kgx.source.source (*module*), 29
kgx.source.sparql_source (*module*), 58
kgx.source.sssom_source (*module*), 44
kgx.source.trapi_source (*module*), 39
kgx.source.tsv_source (*module*), 32
kgx.transformer (*module*), 27
kgx.utils.graph_utils (*module*), 95
kgx.utils.kgx_utils (*module*), 91
kgx.utils.rdf_utils (*module*), 96
kgx.validator (*module*), 70
kgx-graph-summary command line option
    --edge-facet-properties <edge_facet_properties>, 5
    --error-log <error_log>, 5
    --graph-name <graph_name>, 5
    --input-compression <input_compression>, 5
    --input-format <input_format>, 5
    --node-facet-properties <node_facet_properties>, 5
    --output <output>, 5
    --report-format <report_format>, 5
    --report-type <report_type>, 5
    --stream, 5
    -c, 5
    -f, 5
    -i, 5
    -l, 5
    -n, 5
    -o, 5

-r, 5
-s, 5
INPUTS, 6
kgx-merge command line option
--destination <destination>, 6
--merge-config <merge_config>, 6
--processes <processes>, 6
--source <source>, 6
-p, 6
kgx-neo4j-download command line option
--edge-filters <edge_filters>, 7
--node-filters <node_filters>, 7
--output <output>, 6
--output-compression <output_compression>, 6
--output-format <output_format>, 6
--password <password>, 6
--stream, 6
--uri <uri>, 6
--username <username>, 6
-d, 6
-e, 7
-f, 6
-l, 6
-n, 7
-o, 6
-p, 6
-s, 6
-u, 6
kgx-neo4j-upload command line option
--edge-filters <edge_filters>, 7
--input-compression <input_compression>, 7
--input-format <input_format>, 7
--node-filters <node_filters>, 7
--password <password>, 7
--stream, 7
--uri <uri>, 7
--username <username>, 7
-c, 7
-e, 7
-i, 7
-l, 7
-n, 7
-p, 7
-s, 7
-u, 7
INPUTS, 7
kgx-transform command line option
--edge-filters <edge_filters>, 8
--infores-catalog <infores_catalog>, 8
--input-compression <input_compression>, 8
--input-format <input_format>, 8
--knowledge-sources <knowledge_sources>, 8
--node-filters <node_filters>, 8
--output <output>, 8
--output-compression <output_compression>, 8
--output-format <output_format>, 8
--processes <processes>, 8
--source <source>, 8
--stream, 8
--transform-config <transform_config>, 8
-c, 8
-d, 8
-e, 8
-f, 8
-i, 8
-k, 8
-n, 8
-o, 8
-p, 8
INPUTS, 8
kgx-validate command line option
--biolink-release <biolink_release>, 9
--input-compression <input_compression>, 9
--input-format <input_format>, 9
--output <output>, 9
--stream, 9
-b, 9
-c, 9
-i, 9
-o, 9
-s, 9
INPUTS, 9

L

load_edge() (*kgx.source.neo_source.NeoSource method*), 49
load_edge() (*kgx.source.sssom_source.SssomSource method*), 45
load_edge() (*kgx.source.trapi_source.TrapiSource method*), 39
load_edges() (*kgx.source.neo_source.NeoSource method*), 49
load_edges() (*kgx.source.sssom_source.SssomSource method*), 45
load_graph() (*kgx.source.owl_source.OwlSource method*), 56
load_node() (*kgx.source.neo_source.NeoSource method*), 49

load_node() (*kgx.source.sssom_source.SssomSource method*), 45
load_node() (*kgx.source.trapi_source.TrapiSource method*), 40
load_nodes() (*kgx.source.neo_source.NeoSource method*), 49
load_ontologies() (*kgx.curie_lookup_service.CurieLookupService method*), 99

M

merge() (*in module kgx.cli.cli_utils*), 10
merge_all_graphs() (*in module kgx.graph_operations.graph_merge*), 81
merge_edge() (*in module kgx.graph_operations.graph_merge*), 82
merge_graphs() (*in module kgx.graph_operations.graph_merge*), 82
merge_node() (*in module kgx.graph_operations.graph_merge*), 82
MessageLevel (*class in kgx.validator*), 70

N

neo4j_download() (*in module kgx.cli.cli_utils*), 10
neo4j_upload() (*in module kgx.cli.cli_utils*), 11
NeoSink (*class in kgx.sink.neo_sink*), 67
NeoSource (*class in kgx.source.neo_source*), 47
nodes() (*kgx.graph.base_graph.BaseGraph method*), 18
nodes() (*kgx.graph.nx_graph.NxGraph method*), 22
nodes_iter() (*kgx.graph.base_graph.BaseGraph method*), 18
nodes_iter() (*kgx.graph.nx_graph.NxGraph method*), 22
number_of_edges() (*kgx.graph.base_graph.BaseGraph method*), 18
number_of_edges() (*kgx.graph.nx_graph.NxGraph method*), 22
number_of_nodes() (*kgx.graph.base_graph.BaseGraph method*), 18
number_of_nodes() (*kgx.graph.nx_graph.NxGraph method*), 23
NxGraph (*class in kgx.graph.nx_graph*), 20

O

ObographSource (*class in kgx.source.obograph_source*), 42
out_edges() (*kgx.graph.base_graph.BaseGraph method*), 18
out_edges() (*kgx.graph.nx_graph.NxGraph method*), 23
OwlSource (*class in kgx.source.owl_source*), 55

P

parse() (*kgx.source.graph_source.GraphSource method*), 31
parse() (*kgx.source.json_source.JsonSource method*), 35
parse() (*kgx.source.jsonl_source.JsonlSource method*), 37
parse() (*kgx.source.neo_source.NeoSource method*), 49
parse() (*kgx.source.obograph_source.ObographSource method*), 42
parse() (*kgx.source.owl_source.OwlSource method*), 56
parse() (*kgx.source.rdf_source.RdfSource method*), 52
parse() (*kgx.source.sparql_source.SparqlSource method*), 60
parse() (*kgx.source.sssom_source.SssomSource method*), 45
parse() (*kgx.source.trapi_source.TrapiSource method*), 40
parse() (*kgx.source.tsv_source.TsvSource method*), 32
parse_header() (*kgx.source.sssom_source.SssomSource method*), 45
parse_meta() (*kgx.source.obograph_source.ObographSource method*), 42
parse_source() (*in module kgx.cli.cli_utils*), 11
PrefixManager (*class in kgx.prefix_manager*), 97
prepare_data_dict() (*in module kgx.utils.kgx_utils*), 94
prepare_input_args() (*in module kgx.cli.cli_utils*), 12
prepare_output_args() (*in module kgx.cli.cli_utils*), 12
prepare_top_level_args() (*in module kgx.cli.cli_utils*), 13
process() (*kgx.transformer.Transformer method*), 27
process_predicate() (*in module kgx.utils.rdf_utils*), 97
process_predicate() (*kgx.sink.rdf_sink.RdfSink method*), 68
process_predicate() (*kgx.source.owl_source.OwlSource method*), 56
process_predicate() (*kgx.source.rdf_source.RdfSource method*), 53
process_predicate() (*kgx.source.sparql_source.SparqlSource method*), 60

R

RdfSink (*class in kgx.sink.rdf_sink*), 68
RdfSource (*class in kgx.source.rdf_source*), 51

read_edge() (*kgx.source.json_source.JsonSource method*), 35
read_edge() (*kgx.source.jsonl_source.JsonlSource method*), 37
read_edge() (*kgx.source.obograph_source.ObographSource method*), 43
read_edge() (*kgx.source.trapi_source.TrapiSource method*), 40
read_edge() (*kgx.source.tsv_source.TsvSource method*), 33
read_edges() (*kgx.source.graph_source.GraphSource method*), 31
read_edges() (*kgx.source.json_source.JsonSource method*), 35
read_edges() (*kgx.source.jsonl_source.JsonlSource method*), 37
read_edges() (*kgx.source.obograph_source.ObographSource method*), 43
read_edges() (*kgx.source.trapi_source.TrapiSource method*), 40
read_edges() (*kgx.source.tsv_source.TsvSource method*), 33
read_node() (*kgx.source.json_source.JsonSource method*), 35
read_node() (*kgx.source.jsonl_source.JsonlSource method*), 38
read_node() (*kgx.source.obograph_source.ObographSource method*), 43
read_node() (*kgx.source.trapi_source.TrapiSource method*), 40
read_node() (*kgx.source.tsv_source.TsvSource method*), 33
read_nodes() (*kgx.source.graph_source.GraphSource method*), 31
read_nodes() (*kgx.source.json_source.JsonSource method*), 35
read_nodes() (*kgx.source.jsonl_source.JsonlSource method*), 38
read_nodes() (*kgx.source.obograph_source.ObographSource method*), 43
read_nodes() (*kgx.source.trapi_source.TrapiSource method*), 40
read_nodes() (*kgx.source.tsv_source.TsvSource method*), 33
reify() (*kgx.sink.rdf_sink.RdfSink method*), 69
relabel_nodes() (*kgx.graph.base_graph.BaseGraph static method*), 18
relabel_nodes() (*kgx.graph.nx_graph.NxGraph static method*), 23
remap_edge_property() (*in module kgx.graph_operations*), 89
remap_node_identifier() (*in module kgx.graph_operations*), 88
remap_node_property() (*in module kgx.graph_operations*), 88
remove_edge() (*kgx.graph.base_graph.BaseGraph method*), 18
remove_edge() (*kgx.graph.nx_graph.NxGraph method*), 23
remove_node() (*kgx.graph.base_graph.BaseGraph method*), 19
remove_node() (*kgx.graph.nx_graph.NxGraph method*), 23
remove_null() (*in module kgx.utils.kgx_utils*), 94
remove_singleton_nodes() (*in module kgx.graph_operations*), 90
report() (*kgx.validator.Validator static method*), 72

S

sanitize_category() (*kgx.sink.neo_sink.NeoSink static method*), 67
sanitize_import() (*in module kgx.utils.kgx_utils*), 94
save() (*kgx.graph_operations.summarize_graph.GraphSummary method*), 86
save() (*kgx.transformer.Transformer method*), 27
sentencecase_to_camelcase() (*in module kgx.utils.kgx_utils*), 94
sentencecase_to_snakecase() (*in module kgx.utils.kgx_utils*), 94
set_edge_attributes() (*kgx.graph.base_graph.BaseGraph static method*), 19
set_edge_attributes() (*kgx.graph.nx_graph.NxGraph static method*), 23
set_edge_filter() (*kgx.source.graph_source.GraphSource method*), 31
set_edge_filter() (*kgx.source.json_source.JsonSource method*), 35
set_edge_filter() (*kgx.source.jsonl_source.JsonlSource method*), 38
set_edge_filter() (*kgx.source.neo_source.NeoSource method*), 50
set_edge_filter() (*kgx.source.obograph_source.ObographSource method*), 43
set_edge_filter() (*kgx.source.owl_source.OwlSource method*), 57
set_edge_filter() (*kgx.source.rdf_source.RdfSource method*), 53

set_edge_filter() (*kgx.source.source.Source method*), 29
set_edge_filter() (*kgx.source.sparql_source.SparqlSource method*), 60
set_edge_filter() (*kgx.source.sssom_source.SssomSource method*), 46
set_edge_filter() (*kgx.source.trapi_source.TrapiSource method*), 41
set_edge_filter() (*kgx.source.tsv_source.TsvSource method*), 33
set_edge_filters() (*kgx.source.graph_source.GraphSource method*), 31
set_edge_filters() (*kgx.source.json_source.JsonSource method*), 36
set_edge_filters() (*kgx.source.jsonl_source.JsonlSource method*), 38
set_edge_filters() (*kgx.source.neo_source.NeoSource method*), 50
set_edge_filters() (*kgx.source.obograph_source.ObographSource method*), 44
set_edge_filters() (*kgx.source.owl_source.OwlSource method*), 57
set_edge_filters() (*kgx.source.rdf_source.RdfSource method*), 53
set_edge_filters() (*kgx.source.source.Source method*), 29
set_edge_filters() (*kgx.source.sparql_source.SparqlSource method*), 61
set_edge_filters() (*kgx.source.sssom_source.SssomSource method*), 46
set_edge_filters() (*kgx.source.trapi_source.TrapiSource method*), 41
set_edge_filters() (*kgx.source.tsv_source.TsvSource method*), 33
set_edge_properties() (*kgx.sink.trapi_sink.TrapiSink method*), 66
set_edge_properties() (*kgx.sink.tsv_sink.TsvSink method*), 64
set_edge_provenance() (*kgx.source.graph_source.GraphSource method*), 31
set_edge_provenance() (*kgx.source.json_source.JsonSource method*), 36
set_edge_provenance() (*kgx.source.jsonl_source.JsonlSource method*), 38
set_edge_provenance() (*kgx.source.neo_source.NeoSource method*), 50
set_edge_provenance() (*kgx.source.obograph_source.ObographSource method*), 44
set_edge_provenance() (*kgx.source.owl_source.OwlSource method*), 57
set_edge_provenance() (*kgx.source.rdf_source.RdfSource method*), 53
set_edge_provenance() (*kgx.source.source.Source method*), 29
set_edge_provenance() (*kgx.source.sparql_source.SparqlSource method*), 61
set_edge_provenance() (*kgx.source.sssom_source.SssomSource method*), 46
set_edge_provenance() (*kgx.source.trapi_source.TrapiSource method*), 41
set_edge_provenance() (*kgx.source.tsv_source.TsvSource method*), 34
set_node_attributes() (*kgx.graph.base_graph.BaseGraph static method*), 19
set_node_attributes() (*kgx.graph.nx_graph.NxGraph static method*), 23
set_node_filter() (*kgx.source.graph_source.GraphSource method*), 31
set_node_filter() (*kgx.source.json_source.JsonSource method*), 36
set_node_filter() (*kgx.source.jsonl_source.JsonlSource method*), 38
set_node_filter() (*kgx.source.neo_source.NeoSource method*), 50
set_node_filter() (*kgx.source.obograph_source.ObographSource*

method), 44
set_node_filter() (*kgx.source.owl_source.OwlSource method*), 57
set_node_filter() (*kgx.source.rdf_source.RdfSource method*), 53
set_node_filter() (*kgx.source.source.Source method*), 29
set_node_filter() (*kgx.source.sparql_source.SparqlSource method*), 61
set_node_filter() (*kgx.source.sssom_source.SssomSource method*), 46
set_node_filter() (*kgx.source.trapi_source.TrapiSource method*), 41
set_node_filter() (*kgx.source.tsv_source.TsvSource method*), 34
set_node_filters() (*kgx.source.graph_source.GraphSource method*), 32
set_node_filters() (*kgx.source.json_source.JsonSource method*), 36
set_node_filters() (*kgx.source.jsonl_source.JsonlSource method*), 38
set_node_filters() (*kgx.source.neo_source.NeoSource method*), 50
set_node_filters() (*kgx.source.obograph_source.ObographSource method*), 44
set_node_filters() (*kgx.source.owl_source.OwlSource method*), 57
set_node_filters() (*kgx.source.rdf_source.RdfSource method*), 53
set_node_filters() (*kgx.source.source.Source method*), 30
set_node_filters() (*kgx.source.sparql_source.SparqlSource method*), 61
set_node_filters() (*kgx.source.sssom_source.SssomSource method*), 46
set_node_filters() (*kgx.source.trapi_source.TrapiSource method*), 41
set_node_filters() (*kgx.source.tsv_source.TsvSource method*), 34
set_node_properties() (*kgx.sink.trapi_sink.TrapiSink method*), 66
set_node_properties() (*kgx.sink.tsv_sink.TsvSink method*), 64
set_node_property_predicates() (*kgx.source.owl_source.OwlSource method*), 57
set_node_property_predicates() (*kgx.source.rdf_source.RdfSource method*), 53
set_node_property_predicates() (*kgx.source.sparql_source.SparqlSource method*), 61
set_node_provenance() (*kgx.source.graph_source.GraphSource method*), 32
set_node_provenance() (*kgx.source.json_source.JsonSource method*), 36
set_node_provenance() (*kgx.source.jsonl_source.JsonlSource method*), 39
set_node_provenance() (*kgx.source.neo_source.NeoSource method*), 50
set_node_provenance() (*kgx.source.obograph_source.ObographSource method*), 44
set_node_provenance() (*kgx.source.owl_source.OwlSource method*), 57
set_node_provenance() (*kgx.source.rdf_source.RdfSource method*), 54
set_node_provenance() (*kgx.source.source.Source method*), 30
set_node_provenance() (*kgx.source.sparql_source.SparqlSource method*), 61
set_node_provenance() (*kgx.source.sssom_source.SssomSource method*), 46
set_node_provenance() (*kgx.source.trapi_source.TrapiSource method*), 41
set_node_provenance() (*kgx.source.tsv_source.TsvSource method*), 34
set_predicate_mapping() (*kgx.source.owl_source.OwlSource method*), 57
set_predicate_mapping()

(*kgx.source.rdf_source.RdfSource method*), 54
set_predicate_mapping() (*kgx.source.sparql_source.SparqlSource method*), 61
set_prefix_map() (*kgx.prefix_manager.PrefixManager method*), 98
set_prefix_map() (*kgx.source.graph_source.GraphSource method*), 32
set_prefix_map() (*kgx.source.json_source.JsonSource method*), 36
set_prefix_map() (*kgx.source.jsonl_source.JsonlSource method*), 39
set_prefix_map() (*kgx.source.neo_source.NeoSource method*), 50
set_prefix_map() (*kgx.source.obograph_source.ObographSource method*), 44
set_prefix_map() (*kgx.source.owl_source.OwlSource method*), 58
set_prefix_map() (*kgx.source.rdf_source.RdfSource method*), 54
set_prefix_map() (*kgx.source.source.Source method*), 30
set_prefix_map() (*kgx.source.sparql_source.SparqlSource method*), 61
set_prefix_map() (*kgx.source.sssom_source.SssomSource method*), 46
set_prefix_map() (*kgx.source.trapi_source.TrapiSource method*), 41
set_prefix_map() (*kgx.source.tsv_source.TsvSource method*), 34
set_property_types() (*kgx.sink.rdf_sink.RdfSink method*), 69
set_provenance_map() (*kgx.source.graph_source.GraphSource method*), 32
set_provenance_map() (*kgx.source.json_source.JsonSource method*), 36
set_provenance_map() (*kgx.source.jsonl_source.JsonlSource method*), 39
set_provenance_map() (*kgx.source.neo_source.NeoSource method*), 50
set_provenance_map() (*kgx.source.obograph_source.ObographSource method*), 44
set_provenance_map() (*kgx.source.owl_source.OwlSource method*), 58
set_provenance_map() (*kgx.source.rdf_source.RdfSource method*), 54
set_provenance_map() (*kgx.source.source.Source method*), 30
set_provenance_map() (*kgx.source.sparql_source.SparqlSource method*), 61
set_provenance_map() (*kgx.source.sssom_source.SssomSource method*), 46
set_provenance_map() (*kgx.source.trapi_source.TrapiSource method*), 41
set_provenance_map() (*kgx.source.tsv_source.TsvSource method*), 34
set_reverse_predicate_mapping() (*kgx.sink.rdf_sink.RdfSink method*), 69
set_reverse_prefix_map() (*kgx.sink.graph_sink.GraphSink method*), 64
set_reverse_prefix_map() (*kgx.sink.json_sink.JsonSink method*), 65
set_reverse_prefix_map() (*kgx.sink.jsonl_sink.JsonlSink method*), 66
set_reverse_prefix_map() (*kgx.sink.neo_sink.NeoSink method*), 68
set_reverse_prefix_map() (*kgx.sink.rdf_sink.RdfSink method*), 69
set_reverse_prefix_map() (*kgx.sink.sink.Sink method*), 63
set_reverse_prefix_map() (*kgx.sink.trapi_sink.TrapiSink method*), 66
set_reverse_prefix_map() (*kgx.sink.tsv_sink.TsvSink method*), 64
set_reverse_prefix_map() (*kgx.source.json_source.JsonSource method*), 36
set_reverse_prefix_map() (*kgx.source.jsonl_source.JsonlSource method*), 39
set_reverse_prefix_map() (*kgx.source.obograph_source.ObographSource method*), 44
set_reverse_prefix_map() (*kgx.source.sssom_source.SssomSource method*), 47
set_reverse_prefix_map() (*kgx.source.trapi_source.TrapiSource method*), 41
set_reverse_prefix_map() (*kgx.source.tsv_source.TsvSource method*), 34
Sink (*class in kgx.sink.sink*), 63
snakecase_to_sentencecase() (*in module kgx.utils.kgx_utils*), 95

sort_categories() (*in module kgx.graph_operations.clique_merge*), 79
Source (*class in kgx.source.source*), 29
SparqlSource (*class in kgx.source.sparql_source*), 58
SssomSource (*class in kgx.source.sssom_source*), 44
summarize_graph() (*in module kgx.graph_operations.summarize_graph*), 87
summarize_graph() (*kgx.graph_operations.summarize_graph.GraphSummary method*), 86
summarize_graph_edges() (*kgx.graph_operations.summarize_graph.GraphSummary method*), 86
summarize_graph_nodes() (*kgx.graph_operations.summarize_graph.GraphSummary method*), 86

T

transform() (*in module kgx.cli.cli_utils*), 13
transform() (*kgx.transformer.Transformer method*), 27
transform_source() (*in module kgx.cli.cli_utils*), 13
Transformer (*class in kgx.transformer*), 27
TrapiSink (*class in kgx.sink.trapi_sink*), 66
TrapiSource (*class in kgx.source.trapi_source*), 39
triple() (*kgx.source.owl_source.OwlSource method*), 58
triple() (*kgx.source.rdf_source.RdfSource method*), 54
triple() (*kgx.source.sparql_source.SparqlSource method*), 61
TsvSink (*class in kgx.sink.tsv_sink*), 64
TsvSource (*class in kgx.source.tsv_source*), 32

U

unfold_node_property() (*in module kgx.graph_operations*), 90
update_edge() (*kgx.source.owl_source.OwlSource method*), 58
update_edge() (*kgx.source.rdf_source.RdfSource method*), 54
update_edge() (*kgx.source.sparql_source.SparqlSource method*), 62
update_edge_attribute() (*kgx.graph.base_graph.BaseGraph method*), 19
update_edge_attribute() (*kgx.graph.nx_graph.NxGraph method*), 24
update_node() (*kgx.source.owl_source.OwlSource method*), 58
update_node() (*kgx.source.rdf_source.RdfSource method*), 54
update_node() (*kgx.source.sparql_source.SparqlSource method*), 62
update_node_attribute() (*kgx.graph.base_graph.BaseGraph method*), 19
update_node_attribute() (*kgx.graph.nx_graph.NxGraph method*), 24
update_node_categories() (*in module kgx.graph_operations.clique_merge*), 79
update_prefix_map() (*kgx.prefix_manager.PrefixManager method*), 98
update_reverse_prefix_map() (*kgx.prefix_manager.PrefixManager method*), 99
uriref() (*kgx.sink.rdf_sink.RdfSink method*), 69

V

validate() (*in module kgx.cli.cli_utils*), 14
validate() (*kgx.validator.Validator method*), 72
validate_categories() (*kgx.validator.Validator static method*), 72
validate_edge() (*in module kgx.utils.kgx_utils*), 95
validate_edge_predicate() (*kgx.validator.Validator static method*), 72
validate_edge_properties() (*kgx.validator.Validator static method*), 72
validate_edge_property_types() (*kgx.validator.Validator static method*), 73
validate_edge_property_values() (*kgx.validator.Validator static method*), 73
validate_edges() (*kgx.validator.Validator method*), 73
validate_node() (*in module kgx.utils.kgx_utils*), 95
validate_node_properties() (*kgx.validator.Validator static method*), 73
validate_node_property_types() (*kgx.validator.Validator static method*), 74
validate_node_property_values() (*kgx.validator.Validator static method*), 74
validate_nodes() (*kgx.validator.Validator method*), 74
ValidationError (*class in kgx.validator*), 70
Validator (*class in kgx.validator*), 70

W

write_edge() (*kgx.sink.graph_sink.GraphSink method*), 64
write_edge() (*kgx.sink.json_sink.JsonSink method*), 65

write_edge() (*kgx.sink.jsonl_sink.JsonlSink method*), 66
write_edge() (*kgx.sink.neo_sink.NeoSink method*), 68
write_edge() (*kgx.sink.rdf_sink.RdfSink method*), 69
write_edge() (*kgx.sink.sink.Sink method*), 63
write_edge() (*kgx.sink.trapi_sink.TrapiSink method*), 66
write_edge() (*kgx.sink.tsv_sink.TsvSink method*), 64
write_node() (*kgx.sink.graph_sink.GraphSink method*), 64
write_node() (*kgx.sink.json_sink.JsonSink method*), 65
write_node() (*kgx.sink.jsonl_sink.JsonlSink method*), 66
write_node() (*kgx.sink.neo_sink.NeoSink method*), 68
write_node() (*kgx.sink.rdf_sink.RdfSink method*), 69
write_node() (*kgx.sink.sink.Sink method*), 63
write_node() (*kgx.sink.trapi_sink.TrapiSink method*), 66
write_node() (*kgx.sink.tsv_sink.TsvSink method*), 64
write_report() (*kgx.validator.Validator method*), 74