KGX Documentation

Release 0.0.1

Chris Mungall, Deepak Unni, Kenneth Bruskiewicz, Lance Hannes

Jan 09, 2020

CONTENTS

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), represented internally in Python using a networkx MultiDiGraph.

KGX supports Neo4j and RDF triple stores, along with other serialization formats such as TSV, CSV, JSON and TTL.

CHAPTER

ONE

CONTENTS

1.1 Installation

The installation for requires Python 3.6 or greater.

1.1.1 Installation for users

First clone the GitHub repository and then install,

```
git clone https://github.com/NCATS-Tangerine/kgx
cd kgx
python setup.py install
```

1.1.2 Installation for developers

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

```
git clone https://github.com/NCATS-Tangerine/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 Python 3 to create a lightweight virtual environment:

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

pip install -r requirements.txt
```

1.2 Documentation

1.2.1 Transformers

Transformers are classes in KGX that allow for you to

Transformer

The base class for all Transformers in KGX.

class kgx.transformers.transformer.**Transformer**(*source_graph: networkx.classes.multidigraph.MultiDiGraph = None*)

Bases: object

Base class for performing a transformation.

This can be,

- from a source to an in-memory property graph (networkx.MultiDiGraph)
- from an in-memory property graph to a target format or database (Neo4j, CSV, RDF Triple Store, TTL)

categorize()
Find and validate category for every node in self.graph

static dump(*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file(*g: networkx.classes.multidigraph.MultiDiGraph*, *filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

merge_graphs(*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None
Merge all graphs with self.graph

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property

- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

remap_edge_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier (*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report () → None
Print a summary report about self.graph

static restore (*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file (*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

set_filter (*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

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

static 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

static 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

NeoTransformer

class kgx.transformers.neo_transformer.**NeoTransformer**(*graph: networkx.classes.multidigraph.MultiDiGraph = None*, *uri: str = None*, *username: str = None*, *password: str = None*)
Bases: *kgx.transformers.transformer.Transformer*

Transformer for reading from and writing to a Neo4j database.

__init__(*graph: networkx.classes.multidigraph.MultiDiGraph = None*, *uri: str = None*, *username: str = None*, *password: str = None*)
Initialize an instance of NeoTransformer.

categorize()
Find and validate category for every node in self.graph

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

create_constraints(*categories: set*) → None
Create a unique constraint on node 'id' for all `categories` in Neo4j.

Parameters **categories** (*set*) – Set of categories

static dump(*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file(*g: networkx.classes.multidigraph.MultiDiGraph*, *filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

generate_unwind_edge_query(*edge_label: str*) → str
Generate UNWIND cypher query for saving edges into Neo4j.

Query uses `self.DEFAULT_NODE_LABEL` to quickly lookup the required subject and object node.

Parameters **edge_label** (*str*) – Edge label as string

Returns The UNWIND cypher query

Return type str

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_LABEL`. The query uses `self.DEFAULT_NODE_LABEL` as the node label to increase speed for adding nodes. The query also sets label to `self.DEFAULT_NODE_LABEL` for any node to make sure that the CONSTRAINT applies.

Parameters **category** (*str*) – Node category

Returns The UNWIND cypher query

Return type str

get_edges(*skip: int = 0*, *limit: int = 0*, *is_directed: bool = True*) → List[Tuple[neo4jrestclient.client.Node, neo4jrestclient.client.Relationship, neo4jrestclient.client.Node]]
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)

Returns A list of 3-tuples of the form (neo4jrestclient.client.Node, neo4jrestclient.client.Relationship, neo4jrestclient.client.Node)

Return type list

get_filter(*key: str*) → str
Get the value for filter as defined by `key`. This is used as a convenience method for generating cypher queries.

Parameters **key** (*str*) – Name of the filter

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

Return type str

get_nodes(*skip: int = 0*, *limit: int = 0*) → List[neo4jrestclient.client.Node]
Get a page of nodes from the Neo4j database.

Parameters

- **skip** (*int*) – Records to skip

- **limit** (*int*) – Total number of records to query for

Returns A list of neo4jrestclient.client.Node records

Return type list

get_pages(*query_function*, *start: int = 0*, *end: int = None*, *page_size: int = 10000*, ***kwargs*) → list
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** (*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 list

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

load(*start: int = 0*, *end: int = None*, *is_directed: bool = True*) → None
Read nodes and edges from a Neo4j database and create a networkx.MultiDiGraph

Parameters

- **start** (*int*) – Start for pagination
- **end** (*int*) – End for pagination
- **is_directed** (*bool*) – Are edges directed or undirected (`True`, by default, since edges in most cases are directed)

load_edge(*edge: neo4jrestclient.client.Relationship*) → None
Load an edge from neo4jrestclient.client.Relationship into networkx.MultiDiGraph

Parameters **edge** (*neo4jrestclient.client.Relationship*) – An edge

load_edges(*edges: List*) → None
Load edges into networkx.MultiDiGraph

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

load_node(*node: neo4jrestclient.client.Node*) → None
Load node from neo4jrestclient.client.Node into networkx.MultiDiGraph

Parameters **node** (*neo4jrestclient.client.Node*) – A node

load_nodes(*nodes: List[neo4jrestclient.client.Node]*) → None
Load nodes into networkx.MultiDiGraph

Parameters **nodes** (*List[neo4jrestclient.client.Node]*) – A list of node records

merge_graphs (*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None
Merge all graphs with `self.graph`

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property
- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

neo4j_report () → None
Give a summary on the number of nodes and edges in the Neo4j database.

remap_edge_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier (*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report () → None
Print a summary report about self.graph

static restore (*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file(*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

save() → None
Save all nodes and edges from networkx.MultiDiGraph into Neo4j.

TODO: To be deprecated.

save_edge(*obj: dict*) → None
Load an edge into Neo4j.

TODO: To be deprecated.

Parameters **obj** (*dict*) – A dictionary that represents an edge and its properties. The edge must have 'subject', 'edge_label' and 'object' properties. For all other necessary properties, refer to the BioLink Model.

save_edge_unwind(*edges_by_edge_label: Dict[str, list]*) → None
Save all edges into Neo4j using the UNWIND cypher clause.

Parameters **edges_by_edge_label** (*dict*) – A dictionary where edge label is the key and the value is a list of edges with that edge label

save_node(*obj: dict*) → None
Load a node into Neo4j.

TODO: To be deprecated.

Parameters **obj** (*dict*) – A dictionary that represents a node and its properties. The node must have 'id' property. For all other necessary properties, refer to the BioLink Model.

save_node_unwind(*nodes_by_category: Dict[str, list]*) → None
Save all nodes into Neo4j using the UNWIND cypher clause.

Parameters **nodes_by_category** (*Dict[str, list]*) – A dictionary where node category is the key and the value is a list of nodes of that category

save_with_unwind() → None
Save all nodes and edges from networkx.MultiDiGraph into Neo4j using the UNWIND cypher clause.

set_filter(*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

static 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

static 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

PandasTransformer

class kgx.transformers.pandas_transformer.**PandasTransformer**(*source_graph: networkx.classes.multidigraph.MultiDiGraph = None*)
Bases: *kgx.transformers.transformer.Transformer*

Transformer that parses a pandas.DataFrame, and loads nodes and edges into a networkx.MultiDiGraph

categorize()
Find and validate category for every node in self.graph

static dump(*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file(*g: networkx.classes.multidigraph.MultiDiGraph*, *filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

export_edges() → pandas.core.frame.DataFrame
Export edges from networkx.MultiDiGraph as a pandas.DataFrame

Returns A Dataframe where each record corresponds to an edge from the networkx.MultiDiGraph

Return type pandas.DataFrame

export_nodes() → pandas.core.frame.DataFrame
Export nodes from networkx.MultiDiGraph as a pandas.DataFrame

Returns A Dataframe where each record corresponds to a node from the networkx.MultiDiGraph

Return type pandas.DataFrame

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

load(*df: pandas.core.frame.DataFrame*) → None
Load a panda.DataFrame, containing either nodes or edges, into a networkx.MultiDiGraph

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

load_edge (*edge: Dict*) → None
Load an edge into a networkx.MultiDiGraph

Parameters **edge** (*dict*) – An edge

load_edges (*df: pandas.core.frame.DataFrame*) → None
Load edges from pandas.DataFrame into a networkx.MultiDiGraph

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

load_node (*node: Dict*) → None
Load a node into a networkx.MultiDiGraph

Parameters **node** (*dict*) – A node

load_nodes (*df: pandas.core.frame.DataFrame*) → None
Load nodes from pandas.DataFrame into a networkx.MultiDiGraph

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

merge_graphs (*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None
Merge all graphs with `self.graph`

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property
- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

parse (*filename: str*, *input_format: str = 'csv'*, *provided_by: str = None*, ***kwargs*) → None
Parse a CSV/TSV (or plain text) file.

The file can represent either nodes (nodes.csv) or edges (edges.csv) or both (data.tar), where the tar archive contains nodes.csv and edges.csv

The file can also be data.tar.gz or data.tar.bz2

Parameters

- **filename** (*str*) – File to read from
- **input_format** (*str*) – The input file format (`csv`, by default)
- **provided_by** (*str*) – Define the source providing the input file
- **kwargs** (*Dict*) – Any additional arguments

remap_edge_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier (*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report () → None
Print a summary report about self.graph

static restore (*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file (*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

save (*filename: str*, *extension: str = 'csv'*, *mode: str = 'w'*, ***kwargs*) → str
Writes two files representing the node set and edge set of a networkx.MultiDiGraph, and add them to a *.tar* archive.

Parameters

- **filename** (*str*) – Name of tar archive file to create
- **extension** (*str*) – The output file format (`csv`, by default)
- **mode** (*str*) – Form of compression to use (`w`, by default, signifies no compression)
- **kwargs** (*dict*) – Any additional arguments

set_filter (*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

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

static 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

static 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

JsonTransformer

class kgx.transformers.json_transformer.**JsonTransformer**(*source_graph: networkx.classes.multidigraph.MultiDiGraph = None*)
Bases: *kgx.transformers.pandas_transformer.PandasTransformer*

Transformer that parses a JSON, and loads nodes and edges into a networkx.MultiDiGraph

categorize()
Find and validate category for every node in self.graph

static dump(*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file(*g: networkx.classes.multidigraph.MultiDiGraph*, *filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

export() → Dict
Export networkx.MultiDiGraph as a dictionary.

Returns A dictionary with a list nodes and a list of edges

Return type dict

export_edges() → pandas.core.frame.DataFrame
Export edges from networkx.MultiDiGraph as a pandas.DataFrame

Returns A Dataframe where each record corresponds to an edge from the networkx.MultiDiGraph

Return type pandas.DataFrame

export_nodes() → pandas.core.frame.DataFrame
Export nodes from networkx.MultiDiGraph as a pandas.DataFrame

Returns A Dataframe where each record corresponds to a node from the networkx.MultiDiGraph

Return type pandas.DataFrame

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

load(*obj: Dict[str, List]*) → None
Load a JSON object, containing nodes and edges, into a networkx.MultiDiGraph

Parameters **obj** (*dict*) – JSON Object with all nodes and edges

load_edge(*edge: Dict*) → None
Load an edge into a networkx.MultiDiGraph

Parameters **edge** (*dict*) – An edge

load_edges(*edges: List[Dict]*) → None
Load a list of edges into a networkx.MultiDiGraph

Parameters **edges** (*list*) – List of edges

load_node(*node: Dict*) → None
Load a node into a networkx.MultiDiGraph

Parameters **node** (*dict*) – A node

load_nodes(*nodes: List[Dict]*) → None
Load a list of nodes into a networkx.MultiDiGraph

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

merge_graphs(*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None
Merge all graphs with `self.graph`

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property
- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

parse(*filename: str*, *input_format: str = 'json'*, *provided_by: str = None*, ***kwargs*) → None
Parse a JSON file of the format,

{ "nodes" : [. . .], "edges" : [. . .],

}

Parameters

- **filename** (*str*) – JSON file to read from
- **input_format** (*str*) – The input file format (`json`, by default)
- **provided_by** (*str*) – Define the source providing the input file
- **kwargs** (*dict*) – Any additional arguments

remap_edge_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier (*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report () → None
Print a summary report about self.graph

static restore (*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file (*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

save (*filename: str*, ***kwargs*) → None
Write networkx.MultiDiGraph to a file as JSON.

Parameters

- **filename** (*str*) – Filename to write to
- **kwargs** (*dict*) – Any additional arguments

set_filter(*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

static 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

static 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

LogicTermTransformer

class kgx.transformers.logicterm_transformer.**LogicTermTransformer**(*source: Union[kgx.transformers.transformer net- workx.classes.multidigraph.MultiDi = None, out- put_format=None, **args*)

Bases: *kgx.transformers.transformer.Transformer*

TODO: Motivation for LogicTermTransformer?

categorize()
Find and validate category for every node in self.graph

static dump(*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file(*g: networkx.classes.multidigraph.MultiDiGraph*, *filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

merge_graphs(*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None
Merge all graphs with `self.graph`

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property
- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

remap_edge_property(*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier(*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property(*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report() → None
Print a summary report about self.graph

static restore(*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file(*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

save(*filename: str*, *format='sxpr'*, *zipmode='w'*, ***kwargs*)

set_filter(*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

static 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

static 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

NxTransformer

class kgx.transformers.nx_transformer.**GraphMLTransformer**(*source_graph: networkx.classes.multidigraph.MultiDiGraph = None*)
Bases: *kgx.transformers.nx_transformer.NetworkxTransformer*

I/O for graphml TODO: do we need to support GraphML

categorize()
Find and validate category for every node in self.graph

static dump(*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file(*g: networkx.classes.multidigraph.MultiDiGraph*, *filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

merge_graphs(*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None
Merge all graphs with `self.graph`

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property
- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

remap_edge_property(*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier(*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property(*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report() → None
Print a summary report about self.graph

static restore(*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file(*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

set_filter(*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

static 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

static 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

class kgx.transformers.nx_transformer.**NetworkxTransformer**(*source_graph: networkx.classes.multidigraph.MultiDiGraph = None*)

Bases: *kgx.transformers.transformer.Transformer*

Base class for networkx transforms TODO: use case for this class

categorize()
Find and validate category for every node in self.graph

static dump(*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file(*g: networkx.classes.multidigraph.MultiDiGraph*, *filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

merge_graphs(*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None
Merge all graphs with `self.graph`

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property
- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

remap_edge_property(*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier(*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report () → None
Print a summary report about self.graph

static restore (*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file (*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

set_filter (*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

static 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

static 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

RdfGraphMixin

A mixin for handling operations on RDF-stores.

class kgx.transformers.rdf_graph_mixin.**RdfGraphMixin**(*source_graph: networkx.classes.multidigraph.MultiDiGraph = None*)

Bases: object

A mixin that defines the following methods,

- load_networkx_graph(): template method that all deriving classes should implement
- add_node(): method to add a node from a RDF form to property graph form
- add_node_attribute(): method to add a node attribute from a RDF form to property graph form
- add_edge(): method to add an edge from a RDF form to property graph form
- add_edge_attribute(): method to add an edge attribute from an RDF form to property graph form

add_edge(*subject_iri: rdflib.term.URIRef*, *object_iri: rdflib.term.URIRef*, *predicate_iri: rdflib.term.URIRef*) → Tuple[str, str, str]

This method should be used by all derived classes when adding an edge to the networkx.MultiDiGraph. This ensures that the *subject* and *object* identifiers are CURIEs, and that *edge_label* is in the correct form.

Returns the CURIE identifiers used for the *subject* and *object* in the networkx.MultiDiGraph, and the processed *edge_label*.

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

Returns A 3-nary tuple (of the form subject, object, predicate) that represents the edge

Return type Tuple[str, str, str]

add_edge_attribute(*subject_iri: Union[rdflib.term.URIRef, str], object_iri: rdflib.term.URIRef, predicate_iri: rdflib.term.URIRef, key: str, value: str*) → None

Adds an attribute to an edge, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the nodes in the edge does not exist then they will be created using `subject_iri` and `object_iri`.

If the edge itself does not exist then it will be created using `subject_iri`, `object_iri` and `predicate_iri`.

Parameters

- **subject_iri** (*[rdflib.URIRef, str]*) – The IRI of the subject node of an edge in rdflib.Graph
- **object_iri** (*rdflib.URIRef*) – The IRI of the object node of an edge in rdflib.Graph
- **predicate_iri** (*rdflib.URIRef*) – The IRI of the predicate representing an edge in rdflib.Graph
- **key** (*str*) – The name of the attribute. Can be a rdflib.URIRef or URI string

- **value** (*str*) – The value of the attribute

add_node(*iri: rdflib.term.URIRef*) → str

This method should be used by all derived classes when adding a node to the networkx.MultiDiGraph. This ensures that a node's identifier is a CURIE, and that it's *iri* property is set.

Returns the CURIE identifier for the node in the networkx.MultiDiGraph

Parameters **iri** (*rdflib.URIRef*) – IRI of a node

Returns The CURIE identifier of a node

Return type str

add_node_attribute(*iri: Union[rdflib.term.URIRef, str], key: str, value: str*) → None

Add an attribute to a node, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the node does not exist then it is created using the given `iri`.

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** (*str*) – The value of the attribute

load_networkx_graph(*rdfgraph: rdflib.graph.Graph = None*, *predicates: Set[rdflib.term.URIRef] = None*, ***kwargs*) → None

This method should be overridden and be implemented by the derived class, and should load all desired nodes and edges from rdflib.Graph into networkx.MultiDiGraph

Its preferred that this method does not use the networkx API directly when adding nodes, edges, and their attributes.

Instead, Using the following methods,

- `add_node()`
- `add_node_attribute()`
- `add_edge()`
- `add_edge_attribute()`

to ensure that nodes, edges, and their attributes are added in conformance with the BioLink Model, and that URIRef's are translated into CURIEs or BioLink Model elements whenever appropriate.

Parameters

- **rdfgraph** (*rdflib.Graph*) – Graph containing nodes and edges
- **predicates** (*list*) – A list of rdflib.URIRef representing predicates to be loaded
- **kwargs** (*dict*) – Any additional arguments

RdfTransformer

class kgx.transformers.rdf_transformer.**ObanRdfTransformer**(*source_graph: networkx.classes.multidigraph.MultiDiGraph = None*)

Bases: *kgx.transformers.rdf_transformer.RdfTransformer*

Transformer that parses a 'turtle' file and loads triples, as nodes and edges, into a networkx.MultiDiGraph

This Transformer supports OBAN style of modeling where, - it dereifies OBAN.association triples into a property graph form - it reifies property graph into OBAN.association triples

add_edge(*subject_iri: rdflib.term.URIRef*, *object_iri: rdflib.term.URIRef*, *predicate_iri: rdflib.term.URIRef*) → Tuple[str, str, str]

This method should be used by all derived classes when adding an edge to the networkx.MultiDiGraph. This ensures that the *subject* and *object* identifiers are CURIEs, and that *edge_label* is in the correct form.

Returns the CURIE identifiers used for the *subject* and *object* in the networkx.MultiDiGraph, and the processed *edge_label*.

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

Returns A 3-nary tuple (of the form subject, object, predicate) that represents the edge

Return type Tuple[str, str, str]

add_edge_attribute(*subject_iri: Union[rdflib.term.URIRef, str], object_iri: rdflib.term.URIRef, predicate_iri: rdflib.term.URIRef, key: str, value: str*) → None

Adds an attribute to an edge, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the nodes in the edge does not exist then they will be created using `subject_iri` and `object_iri`.

If the edge itself does not exist then it will be created using `subject_iri`, `object_iri` and `predicate_iri`.

Parameters

- **subject_iri** (*[rdflib.URIRef, str]*) – The IRI of the subject node of an edge in rdflib.Graph
- **object_iri** (*rdflib.URIRef*) – The IRI of the object node of an edge in rdflib.Graph
- **predicate_iri** (*rdflib.URIRef*) – The IRI of the predicate representing an edge in rdflib.Graph
- **key** (*str*) – The name of the attribute. Can be a rdflib.URIRef or URI string
- **value** (*str*) – The value of the attribute

add_node(*iri: rdflib.term.URIRef*) → str

This method should be used by all derived classes when adding a node to the networkx.MultiDiGraph. This ensures that a node's identifier is a CURIE, and that it's *iri* property is set.

Returns the CURIE identifier for the node in the networkx.MultiDiGraph

Parameters **iri** (*rdflib.URIRef*) – IRI of a node

Returns The CURIE identifier of a node

Return type str

add_node_attribute(*iri: Union[rdflib.term.URIRef, str], key: str, value: str*) → None
Add an attribute to a node, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the node does not exist then it is created using the given `iri`.

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** (*str*) – The value of the attribute

add_ontology(*file: str*) → None
Load an ontology OWL into a Rdflib.Graph # TODO: is there better way of pre-loading required ontologies?

categorize()
Find and validate category for every node in self.graph

static dump(*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file(*g: networkx.classes.multidigraph.MultiDiGraph, filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

load_networkx_graph(*rdfgraph: rdflib.graph.Graph = None, predicates: Set[rdflib.term.URIRef] = None, **kwargs*) → None
Walk through the rdflib.Graph and load all triples into networkx.MultiDiGraph

Parameters

- **rdfgraph** (*rdflib.Graph*) – Graph containing nodes and edges
- **predicates** (*list*) – A list of rdflib.URIRef representing predicates to be loaded
- **kwargs** (*dict*) – Any additional arguments

load_node_attributes (*rdfgraph: rdflib.graph.Graph*) → None
This method loads the properties of nodes into networkx.MultiDiGraph As there can be many values for a single key, all properties are lists by default.

This method assumes that `RdfTransformer.load_edges()` has been called, and that all nodes have had their IRI as an attribute.

Parameters **rdfgraph** (*rdflib.Graph*) – Graph containing nodes and edges

merge_graphs (*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None
Merge all graphs with `self.graph`

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property
- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

parse (*filename: str = None*, *input_format: str = None*, *provided_by: str = None*, *predicates: Set[rdflib.term.URIRef] = None*) → None
Parse a file, containing triples, into a rdflib.Graph

The file can be either a 'turtle' file or any other format supported by rdflib.

Parameters

- **filename** (*str*) – File to read from.
- **input_format** (*str*) – The input file format. If `None` is provided then the format is guessed using `rdflib.util.guess_format()`
- **provided_by** (*str*) – Define the source providing the input file.

remap_edge_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier (*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report() → None
Print a summary report about self.graph

static restore(*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file(*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

save(*filename: str = None*, *output_format: str = 'turtle'*, ***kwargs*) → None
Transform networkx.MultiDiGraph into rdflib.Graph that follow OBAN-style reification and export this graph as a file (`turtle`, by default).

Parameters

- **filename** (*str*) – Filename to write to
- **output_format** (*str*) – The output format; default: `turtle`
- **kwargs** (*dict*) – Any additional arguments

save_attribute(*rdfgraph: rdflib.graph.Graph, object_iri: rdflib.term.URIRef, key: str, value: Union[List[str], str]*) → None
Saves a node or edge attributes from networkx.MultiDiGraph into rdflib.Graph

Intended to be used within *ObanRdfTransformer.save()*.

Parameters

- **rdfgraph** (*rdflib.Graph*) – Graph containing nodes and edges
- **object_iri** (*rdflib.URIRef*) – IRI of an object in the graph
- **key** (*str*) – The name of the attribute
- **value** (*Union[List[str], str]*) – The value of the attribute; Can be either a List or just a string

set_filter(*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

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

static 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

static 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

class kgx.transformers.rdf_transformer.**RdfOwlTransformer**(*source_graph: networkx.classes.multidigraph.MultiDiGraph = None*)
Bases: *kgx.transformers.rdf_transformer.RdfTransformer*

Transformer that parses an OWL ontology in RDF, while retaining class-class relationships.

add_edge(*subject_iri: rdflib.term.URIRef*, *object_iri: rdflib.term.URIRef*, *predicate_iri: rdflib.term.URIRef*) → Tuple[str, str, str]
This method should be used by all derived classes when adding an edge to the networkx.MultiDiGraph. This ensures that the *subject* and *object* identifiers are CURIEs, and that *edge_label* is in the correct form.

Returns the CURIE identifiers used for the *subject* and *object* in the networkx.MultiDiGraph, and the processed *edge_label*.

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

Returns A 3-nary tuple (of the form subject, object, predicate) that represents the edge

Return type Tuple[str, str, str]

add_edge_attribute(*subject_iri: Union[rdflib.term.URIRef, str], object_iri: rdflib.term.URIRef, predicate_iri: rdflib.term.URIRef, key: str, value: str*) → None
Adds an attribute to an edge, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the nodes in the edge does not exist then they will be created using `subject_iri` and `object_iri`.

If the edge itself does not exist then it will be created using `subject_iri`, `object_iri` and `predicate_iri`.

Parameters

- **subject_iri** (*[rdflib.URIRef, str]*) – The IRI of the subject node of an edge in rdflib.Graph
- **object_iri** (*rdflib.URIRef*) – The IRI of the object node of an edge in rdflib.Graph
- **predicate_iri** (*rdflib.URIRef*) – The IRI of the predicate representing an edge in rdflib.Graph
- **key** (*str*) – The name of the attribute. Can be a rdflib.URIRef or URI string
- **value** (*str*) – The value of the attribute

add_node (*iri: rdflib.term.URIRef*) → str
This method should be used by all derived classes when adding a node to the networkx.MultiDiGraph. This ensures that a node's identifier is a CURIE, and that it's *iri* property is set.

Returns the CURIE identifier for the node in the networkx.MultiDiGraph

Parameters **iri** (*rdflib.URIRef*) – IRI of a node

Returns The CURIE identifier of a node

Return type str

add_node_attribute (*iri: Union[rdflib.term.URIRef, str], key: str, value: str*) → None
Add an attribute to a node, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the node does not exist then it is created using the given `iri`.

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** (*str*) – The value of the attribute

add_ontology (*file: str*) → None
Load an ontology OWL into a Rdflib.Graph # TODO: is there better way of pre-loading required ontologies?

categorize ()
Find and validate category for every node in self.graph

static dump (*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file (*g: networkx.classes.multidigraph.MultiDiGraph, filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

load_networkx_graph(*rdfgraph: rdflib.graph.Graph = None*, *predicates: Set[rdflib.term.URIRef] = None*, ***kwargs*) → None
Walk through the rdflib.Graph and load all triples into networkx.MultiDiGraph

Parameters

- **rdfgraph** (*rdflib.Graph*) – Graph containing nodes and edges
- **predicates** (*list*) – A list of rdflib.URIRef representing predicates to be loaded
- **kwargs** (*dict*) – Any additional arguments

load_node_attributes(*rdfgraph: rdflib.graph.Graph*) → None
This method loads the properties of nodes into networkx.MultiDiGraph As there can be many values for a single key, all properties are lists by default.

This method assumes that `RdfTransformer.load_edges()` has been called, and that all nodes have had their IRI as an attribute.

Parameters **rdfgraph** (*rdflib.Graph*) – Graph containing nodes and edges

merge_graphs(*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None
Merge all graphs with `self.graph`

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property
- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

parse(*filename: str = None*, *input_format: str = None*, *provided_by: str = None*, *predicates: Set[rdflib.term.URIRef] = None*) → None
Parse a file, containing triples, into a rdflib.Graph

The file can be either a 'turtle' file or any other format supported by rdflib.

Parameters

- **filename** (*str*) – File to read from.
- **input_format** (*str*) – The input file format. If `None` is provided then the format is guessed using `rdflib.util.guess_format()`
- **provided_by** (*str*) – Define the source providing the input file.

remap_edge_property(*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier (*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report () → None
Print a summary report about self.graph

static restore (*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file (*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

set_filter (*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

static 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

static 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

class kgx.transformers.rdf_transformer.**RdfTransformer**(*source_graph: networkx.classes.multidigraph.MultiDiGraph = None*)
Bases: *kgx.transformers.rdf_graph_mixin.RdfGraphMixin*, *kgx.transformers.transformer.Transformer*

Transformer that parses RDF and loads triples, as nodes and edges, into a networkx.MultiDiGraph

This is the base class which is used to implement other RDF-based transformers.

add_edge(*subject_iri: rdflib.term.URIRef*, *object_iri: rdflib.term.URIRef*, *predicate_iri: rdflib.term.URIRef*) → Tuple[str, str, str]
This method should be used by all derived classes when adding an edge to the networkx.MultiDiGraph. This ensures that the *subject* and *object* identifiers are CURIEs, and that *edge_label* is in the correct form.

Returns the CURIE identifiers used for the *subject* and *object* in the networkx.MultiDiGraph, and the processed *edge_label*.

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

Returns A 3-nary tuple (of the form subject, object, predicate) that represents the edge

Return type Tuple[str, str, str]

add_edge_attribute(*subject_iri: Union[rdflib.term.URIRef, str], object_iri: rdflib.term.URIRef, predicate_iri: rdflib.term.URIRef, key: str, value: str*) → None
Adds an attribute to an edge, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the nodes in the edge does not exist then they will be created using `subject_iri` and `object_iri`.

If the edge itself does not exist then it will be created using `subject_iri`, `object_iri` and `predicate_iri`.

Parameters

- **subject_iri** (*[rdflib.URIRef, str]*) – The IRI of the subject node of an edge in rdflib.Graph
- **object_iri** (*rdflib.URIRef*) – The IRI of the object node of an edge in rdflib.Graph
- **predicate_iri** (*rdflib.URIRef*) – The IRI of the predicate representing an edge in rdflib.Graph

- **key** (*str*) – The name of the attribute. Can be a rdflib.URIRef or URI string
- **value** (*str*) – The value of the attribute

add_node(*iri: rdflib.term.URIRef*) → str
This method should be used by all derived classes when adding a node to the networkx.MultiDiGraph. This ensures that a node's identifier is a CURIE, and that it's *iri* property is set.

Returns the CURIE identifier for the node in the networkx.MultiDiGraph

Parameters **iri** (*rdflib.URIRef*) – IRI of a node

Returns The CURIE identifier of a node

Return type str

add_node_attribute(*iri: Union[rdflib.term.URIRef, str], key: str, value: str*) → None
Add an attribute to a node, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the node does not exist then it is created using the given `iri`.

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** (*str*) – The value of the attribute

add_ontology(*file: str*) → None
Load an ontology OWL into a Rdflib.Graph # TODO: is there better way of pre-loading required ontologies?

categorize()
Find and validate category for every node in self.graph

static dump(*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file(*g: networkx.classes.multidigraph.MultiDiGraph*, *filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

load_networkx_graph(*rdfgraph: rdflib.graph.Graph = None*, *predicates: Set[rdflib.term.URIRef] = None*, ***kwargs*) → None
Walk through the rdflib.Graph and load all required triples into networkx.MultiDiGraph

By default this method loads the following predicates,

- `RDFS.subClassOf`
- `OWL.sameAs`
- `OWL.equivalentClass`
- `is_about` (IAO:0000136)
- `has_subsequence` (RO:0002524)
- `is_subsequence_of` (RO:0002525)

This behavior can be overridden by providing a list of rdflib.URIRef that ought to be loaded via the `predicates` parameter.

Parameters

- **rdfgraph** (*rdflib.Graph*) – Graph containing nodes and edges
- **predicates** (*list*) – A list of rdflib.URIRef representing predicates to be loaded
- **kwargs** (*dict*) – Any additional arguments

load_node_attributes (*rdfgraph: rdflib.graph.Graph*) → None

This method loads the properties of nodes into networkx.MultiDiGraph As there can be many values for a single key, all properties are lists by default.

This method assumes that `RdfTransformer.load_edges()` has been called, and that all nodes have had their IRI as an attribute.

Parameters **rdfgraph** (*rdflib.Graph*) – Graph containing nodes and edges

merge_graphs (*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None

Merge all graphs with `self.graph`

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property
- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

parse (*filename: str = None*, *input_format: str = None*, *provided_by: str = None*, *predicates: Set[rdflib.term.URIRef] = None*) → None

Parse a file, containing triples, into a rdflib.Graph

The file can be either a 'turtle' file or any other format supported by rdflib.

Parameters

- **filename** (*str*) – File to read from.
- **input_format** (*str*) – The input file format. If `None` is provided then the format is guessed using `rdflib.util.guess_format()`
- **provided_by** (*str*) – Define the source providing the input file.

remap_edge_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier (*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report () → None
Print a summary report about self.graph

static restore (*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file (*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

set_filter (*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

static 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

static 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

SparqlTransformer

class kgx.transformers.sparql_transformer.**MonarchSparqlTransformer**(*source_graph: networkx.classes.multidigraph.MultiD = None*)

Bases: *kgx.transformers.sparql_transformer.SparqlTransformer*

see neo_transformer for discussion

add_edge(*subject_iri: rdflib.term.URIRef, object_iri: rdflib.term.URIRef, predicate_iri: rdflib.term.URIRef*) → Tuple[str, str, str]
This method should be used by all derived classes when adding an edge to the networkx.MultiDiGraph. This ensures that the *subject* and *object* identifiers are CURIEs, and that *edge_label* is in the correct form.

Returns the CURIE identifiers used for the *subject* and *object* in the networkx.MultiDiGraph, and the processed *edge_label*.

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

Returns A 3-nary tuple (of the form subject, object, predicate) that represents the edge

Return type Tuple[str, str, str]

add_edge_attribute(*subject_iri: Union[rdflib.term.URIRef, str], object_iri: rdflib.term.URIRef, predicate_iri: rdflib.term.URIRef, key: str, value: str*) → None
Adds an attribute to an edge, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the nodes in the edge does not exist then they will be created using `subject_iri` and `object_iri`.

If the edge itself does not exist then it will be created using `subject_iri`, `object_iri` and `predicate_iri`.

Parameters

- **subject_iri** (*[rdflib.URIRef, str]*) – The IRI of the subject node of an edge in rdflib.Graph

- **object_iri** (*rdflib.URIRef*) – The IRI of the object node of an edge in rdflib.Graph
- **predicate_iri** (*rdflib.URIRef*) – The IRI of the predicate representing an edge in rdflib.Graph
- **key** (*str*) – The name of the attribute. Can be a rdflib.URIRef or URI string
- **value** (*str*) – The value of the attribute

add_node(*iri: rdflib.term.URIRef*) → str
This method should be used by all derived classes when adding a node to the networkx.MultiDiGraph. This ensures that a node's identifier is a CURIE, and that it's *iri* property is set.

Returns the CURIE identifier for the node in the networkx.MultiDiGraph

Parameters **iri** (*rdflib.URIRef*) – IRI of a node

Returns The CURIE identifier of a node

Return type str

add_node_attribute(*iri: Union[rdflib.term.URIRef, str], key: str, value: str*) → None
Add an attribute to a node, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the node does not exist then it is created using the given `iri`.

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** (*str*) – The value of the attribute

categorize()
Find and validate category for every node in self.graph

static dump(*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file(*g: networkx.classes.multidigraph.MultiDiGraph, filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

get_filters() → Dict
Gets the current filter map, transforming if necessary.

Returns Returns a dictionary with all filters

Return type dict

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

load_networkx_graph(*rdfgraph: rdflib.graph.Graph = None*, *predicates: Set[rdflib.term.URIRef] = None*, ***kwargs*) → None
Fetch triples from the SPARQL endpoint and load them as edges.

Parameters

- **rdfgraph** (*rdflib.Graph*) – A rdflib Graph (unused)
- **predicates** (*set*) – A set containing predicates in rdflib.URIRef form
- **kwargs** (*dict*) – Any additional arguments.

merge_graphs(*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None
Merge all graphs with `self.graph`

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property
- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

query(*q: str*) → Dict
Query a SPARQL endpoint.

Parameters **q** (*str*) – The query string

Returns A dictionary containing results from the query

Return type dict

remap_edge_property(*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier(*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report () → None
Print a summary report about self.graph

static restore (*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file (*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

set_filter (*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

static 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

static 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

class kgx.transformers.sparql_transformer.**RedSparqlTransformer**(*source_graph: networkx.classes.multidigraph.MultiDiGrap = None, url: str = 'http://graphdb.dumontierlab.com/reposi red-kg'*)

Bases: *kgx.transformers.sparql_transformer.SparqlTransformer*

Transformer for communicating with Data2Services Knowledge Graph, a.k.a. Translator Red KG.

add_edge(*subject_iri: rdflib.term.URIRef, object_iri: rdflib.term.URIRef, predicate_iri: rdflib.term.URIRef*) → Tuple[str, str, str]

This method should be used by all derived classes when adding an edge to the networkx.MultiDiGraph. This ensures that the *subject* and *object* identifiers are CURIEs, and that *edge_label* is in the correct form.

Returns the CURIE identifiers used for the *subject* and *object* in the networkx.MultiDiGraph, and the processed *edge_label*.

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

Returns A 3-nary tuple (of the form subject, object, predicate) that represents the edge

Return type Tuple[str, str, str]

add_edge_attribute(*subject_iri: Union[rdflib.term.URIRef, str], object_iri: rdflib.term.URIRef, predicate_iri: rdflib.term.URIRef, key: str, value: str*) → None

Adds an attribute to an edge, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the nodes in the edge does not exist then they will be created using `subject_iri` and `object_iri`.

If the edge itself does not exist then it will be created using `subject_iri`, `object_iri` and `predicate_iri`.

Parameters

- **subject_iri** (*[rdflib.URIRef, str]*) – The IRI of the subject node of an edge in rdflib.Graph
- **object_iri** (*rdflib.URIRef*) – The IRI of the object node of an edge in rdflib.Graph
- **predicate_iri** (*rdflib.URIRef*) – The IRI of the predicate representing an edge in rdflib.Graph
- **key** (*str*) – The name of the attribute. Can be a rdflib.URIRef or URI string
- **value** (*str*) – The value of the attribute

add_node(*iri: rdflib.term.URIRef*) → str

This method should be used by all derived classes when adding a node to the networkx.MultiDiGraph. This ensures that a node's identifier is a CURIE, and that it's *iri* property is set.

Returns the CURIE identifier for the node in the networkx.MultiDiGraph

Parameters **iri** (*rdflib.URIRef*) – IRI of a node

Returns The CURIE identifier of a node

Return type str

add_node_attribute(*iri: Union[rdflib.term.URIRef, str], key: str, value: str*) → None
Add an attribute to a node, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the node does not exist then it is created using the given `iri`.

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** (*str*) – The value of the attribute

categorize() → None
Checks for a node's category property and assigns a category from BioLink Model. TODO: categorize for edges?

static dump(*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file(*g: networkx.classes.multidigraph.MultiDiGraph*, *filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

get_filters() → Dict
Gets the current filter map, transforming if necessary.

Returns Returns a dictionary with all filters

Return type dict

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

load_networkx_graph(*rdfgraph: rdflib.graph.Graph = None*, *predicates: Set[rdflib.term.URIRef] = None*, ***kwargs: Dict*) → None
Fetch all triples using the specified predicates and add them to networkx.MultiDiGraph.

Parameters

- **rdfgraph** (*rdflib.Graph*) – A rdflib Graph (unused)
- **predicates** (*set*) – A set containing predicates in rdflib.URIRef form

- **kwargs** (*dict*) – Any additional arguments. Ex: specifying 'limit' argument will limit the number of triples fetched.

load_nodes(*node_set: Set*) → None
Load nodes into networkx.MultiDiGraph.

This method queries the SPARQL endpoint for all triples where nodes in the node_set is a subject.

Parameters **node_set** (*list*) – A list of node CURIEs

merge_graphs(*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None
Merge all graphs with `self.graph`

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property
- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

query(*q: str*) → Dict
Query a SPARQL endpoint.

Parameters **q** (*str*) – The query string

Returns A dictionary containing results from the query

Return type dict

remap_edge_property(*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier(*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property(*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report() → None
Print a summary report about self.graph

static restore(*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file(*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

set_filter(*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

static 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

static 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

class kgx.transformers.sparql_transformer.**SparqlTransformer**(*source_graph: networkx.classes.multidigraph.MultiDiGraph = None, url: str = None*)

Bases: *kgx.transformers.rdf_graph_mixin.RdfGraphMixin*, *kgx.transformers.transformer.Transformer*

Transformer for communicating with a SPARQL endpoint.

add_edge(*subject_iri: rdflib.term.URIRef, object_iri: rdflib.term.URIRef, predicate_iri: rdflib.term.URIRef*) → Tuple[str, str, str]

This method should be used by all derived classes when adding an edge to the networkx.MultiDiGraph. This ensures that the *subject* and *object* identifiers are CURIEs, and that *edge_label* is in the correct form.

Returns the CURIE identifiers used for the *subject* and *object* in the networkx.MultiDiGraph, and the processed *edge_label*.

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

Returns A 3-nary tuple (of the form subject, object, predicate) that represents the edge

Return type Tuple[str, str, str]

add_edge_attribute(*subject_iri: Union[rdflib.term.URIRef, str], object_iri: rdflib.term.URIRef, predicate_iri: rdflib.term.URIRef, key: str, value: str*) → None

Adds an attribute to an edge, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the nodes in the edge does not exist then they will be created using `subject_iri` and `object_iri`.

If the edge itself does not exist then it will be created using `subject_iri`, `object_iri` and `predicate_iri`.

Parameters

- **subject_iri** (*[rdflib.URIRef, str]*) – The IRI of the subject node of an edge in rdflib.Graph
- **object_iri** (*rdflib.URIRef*) – The IRI of the object node of an edge in rdflib.Graph
- **predicate_iri** (*rdflib.URIRef*) – The IRI of the predicate representing an edge in rdflib.Graph
- **key** (*str*) – The name of the attribute. Can be a rdflib.URIRef or URI string
- **value** (*str*) – The value of the attribute

add_node(*iri: rdflib.term.URIRef*) → str

This method should be used by all derived classes when adding a node to the networkx.MultiDiGraph. This ensures that a node's identifier is a CURIE, and that it's *iri* property is set.

Returns the CURIE identifier for the node in the networkx.MultiDiGraph

Parameters **iri** (*rdflib.URIRef*) – IRI of a node

Returns The CURIE identifier of a node

Return type str

add_node_attribute(*iri: Union[rdflib.term.URIRef, str], key: str, value: str*) → None

Add an attribute to a node, while taking into account whether the attribute should be multi-valued. Multi-valued properties will not contain duplicates.

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

If the node does not exist then it is created using the given `iri`.

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** (*str*) – The value of the attribute

categorize()
Find and validate category for every node in self.graph

static dump(*g: networkx.classes.multidigraph.MultiDiGraph*) → Dict
Convert networkx.MultiDiGraph as a dictionary.

Parameters **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary

Returns A dictionary

Return type dict

static dump_to_file(*g: networkx.classes.multidigraph.MultiDiGraph*, *filename: str*) → None
Serialize networkx.MultiDiGraph as JSON and write to file.

Parameters

- **g** (*networkx.MultiDiGraph*) – Graph to convert as a dictionary
- **filename** (*str*) – File to write the JSON

get_filters() → Dict
Gets the current filter map, transforming if necessary.

Returns Returns a dictionary with all filters

Return type dict

is_empty() → bool
Check whether self.graph is empty.

Returns A boolean value asserting whether the graph is empty or not

Return type bool

load_networkx_graph(*rdfgraph: rdflib.graph.Graph = None*, *predicates: Set[rdflib.term.URIRef] = None*, ***kwargs*) → None
Fetch triples from the SPARQL endpoint and load them as edges.

Parameters

- **rdfgraph** (*rdflib.Graph*) – A rdflib Graph (unused)
- **predicates** (*set*) – A set containing predicates in rdflib.URIRef form
- **kwargs** (*dict*) – Any additional arguments.

merge_graphs(*graphs: List[networkx.classes.multidigraph.MultiDiGraph]*) → None
Merge all graphs with `self.graph`

- If two nodes with same 'id' exist in two graphs, the nodes will be merged based on the 'id'
- If two nodes with the same 'id' exists in two graphs and they both have conflicting values for a property, then the value is overwritten from left to right
- If two edges with the same 'key' exists in two graphs, the edge will be merged based on the 'key' property

- If two edges with the same 'key' exists in two graphs and they both have one or more conflicting values for a property, then the value is overwritten from left to right

Parameters **graphs** (*List[networkx.MultiDiGraph]*) – List of graphs that are to be merged with self.graph

query (*q: str*) → Dict
Query a SPARQL endpoint.

Parameters **q** (*str*) – The query string

Returns A dictionary containing results from the query

Return type dict

remap_edge_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in edge `old_property` attribute with value from edge `new_property` attribute.

Parameters

- **type** (*string*) – label 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

remap_node_identifier (*type: str*, *new_property: str*, *prefix=None*) → None
Remap a node's 'id' attribute with value from a node's `new_property` attribute.

Parameters

- **type** (*string*) – label referring to nodes whose 'id' needs to be remapped
- **new_property** (*string*) – property name from which the new value is pulled from
- **prefix** (*string*) – signifies that the value for `new_property` is a list and the `prefix` indicates which value to pick from the list

remap_node_property (*type: str*, *old_property: str*, *new_property: str*) → None
Remap the value in node `old_property` attribute with value from node `new_property` attribute.

Parameters

- **type** (*string*) – label 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

report () → None
Print a summary report about self.graph

static restore (*data: Dict*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a dictionary.

Parameters **data** (*dict*) – Dictionary containing nodes and edges

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

static restore_from_file (*filename*) → networkx.classes.multidigraph.MultiDiGraph
Deserialize a networkx.MultiDiGraph from a JSON file.

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

Returns A networkx.MultiDiGraph representation

Return type networkx.MultiDiGraph

set_filter(*key: str, value: Union[List[str], str]*) → None
Set a filter, defined by a key and value pair. These filters are used to reduce the search space.

Parameters

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

static 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

static 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

1.2.2 Operations

This module provides a set of operations that are supported by KGX.

Clique Merge

class kgx.operations.clique_merge.**CliqueMerge**(*prefix_prioritization_map: dict = None*)
Bases: object

build_cliques(*target_graph: networkx.classes.multidigraph.MultiDiGraph*)
Builds a clique graph from same_as edges in target_graph.

Parameters **target_graph** (*networkx.MultiDiGraph*) – A MultiDiGraph that contains nodes and edges

Returns The clique graph with only same_as edges

Return type networkx.Graph

consolidate_edges() → networkx.classes.multidigraph.MultiDiGraph
Move all edges from nodes in a clique to the clique leader.

Returns The target graph where all edges from nodes in a clique are moved to clique leader

Return type nx.MultiDiGraph

elect_leader()
Elect leader for each clique in a graph.

get_category_from_equivalence(*node: str*, *attributes: dict*) → str
Get category for a node based on its equivalent nodes in a graph.

Parameters

- **node** (*str*) – Node identifier
- **attributes** (*dict*) – Node's attributes

Returns Category for the node

Return type str

get_leader_by_annotation(*clique: list*) → Tuple[Optional[str], Optional[str]]
Get leader by searching for leader annotation property in any of the nodes in a given clique.

Parameters **clique** (*list*) – A list of nodes from 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]]

get_leader_by_prefix_priority(*clique: list*, *prefix_priority_list: list*) → Tuple[Optional[str], Optional[str]]
Get leader from clique based on a given prefix priority.

Parameters

- **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]]

get_leader_by_sort(*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 **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]]

get_the_most_specific_category(*categories: list*) → Tuple[str, list]
From a list of categories, it tries to fetch ancestors for all. The category with the longest ancestor is considered to be the most specific.

Parameters **categories** (*list*) – A list of categories

Returns A tuple of the most specific category and a list of ancestors of that category

Return type tuple[str, list]

update_categories(*clique: list*)
For a given clique, get category for each node in clique and validate against BioLink Model, mapping to BioLink Model category where needed.

Ex.: If a node has *gene* as its category, then this method adds all of its ancestors.

Parameters **clique** (*list*) – A list of nodes from a clique

validate_categories(*clique: list*) → Tuple[str, list]
For nodes in a clique, validate the category for each node to make sure that all nodes in a clique are of the same type.

Parameters **clique** (*list*) – A list of nodes from a clique

Returns A tuple of clique category string and a list of invalid nodes

Return type tuple[str, list]

1.2.3 Utilities

The *utilities* module include all the utility methods used throughout KGX.

graph_utils

kgx.utils.graph_utils.**curie_lookup**(*curie: str*) → 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 str

kgx.utils.graph_utils.**get_ancestors**(*graph: networkx.classes.multidigraph.MultiDiGraph*, *node: str*, *relations: List[str] = None*) → List[str]
Return all *ancestors* of specified node, filtered by `relations`.

Parameters

- **graph** (*networkx.MultiDiGraph*) – 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: networkx.classes.multidigraph.MultiDiGraph*, *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** (*networkx.MultiDiGraph*) – 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: networkx.classes.multidigraph.MultiDiGraph*, *node: str*, *relations: List[str] = None*) → List[str]
Return all direct *parents* of a specified node, filtered by `relations`.

Parameters

- **graph** (*networkx.MultiDiGraph*) – Graph to traverse
- **node** (*str*) – node identifier
- **relations** (*List[str]*) – list of relations

Returns A list of parent node(s)

Return type List[str]

kgx_utils

kgx.utils.kgx_utils.**camelcase_to_sentencecase**(*s: str*) → str
Convert CamelCase to sentence case.

Parameters **s** (*str*) – Input string in CamelCase

Returns a normal string

Return type str

kgx.utils.kgx_utils.**contract**(*uri*) → str
Contract a URI a CURIE. We sort the curies to ensure that we take the same item every time.

Parameters **uri** (*Union[rdflib.term.URIRef, str]*) – A URI

Returns The CURIE

Return type str

kgx.utils.kgx_utils.**generate_edge_key**(*s: str*, *edge_label: str*, *o: str*) → str
Generates an edge key based on a given subject, edge_label and object.

Parameters

- **s** (*str*) – Subject
- **edge_label** (*str*) – Edge label
- **o** (*str*) – Object

Returns Edge key as a string

Return type str

kgx.utils.kgx_utils.**get_biolink_mapping**(*category*)
Get a BioLink Model mapping for a given `category`.

Parameters **category** (*str*) – A category for which there is a mapping in BioLink Model

Returns A BioLink Model class corresponding to `category`

Return type str

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_toolkit**() → bmt.Toolkit
Get an instance of bmt.Toolkit If there no instance defined, then one is instantiated and returned.

Returns an instance of bmt.Toolkit

Return type bmt.Toolkit

kgx.utils.kgx_utils.**make_curie**(*uri*) → str
Convert a given URI into a CURIE. This method tries to handle the `http` and `https` ambiguity in URI contraction.

> **Warning:** This is a temporary solution and will be deprecated in the near future.

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 a normal string

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 a normal string

Return type str

model_utils

TODO: add methods for ensuring that other biolink model specifications hold, like that all required properties are present and that they have the correct multiplicity, and that all identifiers are CURIE's.

kgx.utils.model_utils.**make_valid_types**(*G: networkx.classes.multidigraph.MultiDiGraph*) → None
Ensures that all the nodes have valid categories, and that all edges have valid edge labels.

Nodes will be deleted if they have no name and have no valid categories. If a node has no valid category but does have a name then its category will be set to the default category "named thing".

Edges with invalid edge labels will have their edge label set to the default value "related_to"

rdf_utils

kgx.utils.rdf_utils.**infer_category**(*iri: rdflib.term.URIRef*, *rdfgraph: rdflib.graph.Graph*) → List[str]
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 List[str]

kgx.utils.rdf_utils.**process_iri**(*iri: Union[str, rdflib.term.URIRef]*) → str
Casts iri to a string, and then checks whether it maps to any pre-defined values. If so returns that value, otherwise converts that iri to a curie and returns.

Parameters **iri** (*Union[str, URIRef]*) – IRI to process; can be a string or a rdflib.term.URIRef

Returns A string corresponding to the IRI

Return type str

1.2.4 KGX CLI

Knowledge Graph Exchange CLI entrypoint.

```
KGX CLI [OPTIONS] COMMAND [ARGS]...
```

Options

--debug
Prints the stack trace if error occurs

--version
Show the version and exit.

edge-summary

Loads and summarizes a knowledge graph edge set, where the input is a file.

```
KGX CLI edge-summary [OPTIONS] FILEPATH
```

Options

--input-type <input_type>

Options tar|txt|csv|tsv|graphml|ttl|json|rq|owl

-m, --max_rows <max_rows>
The maximum number of rows to return

-o, --output <output>

Arguments

FILEPATH
Required argument

load-and-merge

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

```
KGX CLI load-and-merge [OPTIONS] LOAD_CONFIG
```

Arguments

LOAD_CONFIG
Required argument

neo4j-download

Download nodes and edges from Neo4j database.

```
KGX CLI neo4j-download [OPTIONS]
```

Options

-a, --address <address>
[required]

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

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

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

--output-type <output_type>

Options tar|txt|csv|tsv|graphml|ttl|json|rq|owl

--subject-label <subject_label>

--object-label <object_label>

--edge-label <edge_label>

--directed <directed>
Whether the edges are directed

--stop-after <stop_after>
Once this many edges are downloaded the application will finish

--page-size <page_size>
The size of pages to download for each batch

neo4j-edge-summary

Get a summary of all the edges in a Neo4j database.

```
KGX CLI neo4j-edge-summary [OPTIONS]
```

Options

-a, --address <address>
[required]

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

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

-o, --output <output>

neo4j-node-summary

Get a summary of all the nodes in a Neo4j database.

```
KGX CLI neo4j-node-summary [OPTIONS]
```

Options

-a, --address <address>
[required]

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

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

-o, --output <output>

neo4j-upload

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

```
KGX CLI neo4j-upload [OPTIONS] INPUTS...
```

Options

--input-type <input_type>

Options tar|txt|csv|tsv|graphml|ttl|json|rq|owl

--use-unwind
Loads using UNWIND cypher clause, which is quicker

-a, --address <address>
[required]

-u, --username <username>

-p, --password <password>

Arguments

INPUTS
Required argument(s)

node-summary

Loads and summarizes a knowledge graph node set, where the input is a file.

```
KGX CLI node-summary [OPTIONS] FILEPATH
```

Options

--input-type <input_type>

Options tar|txt|csv|tsv|graphml|ttl|json|rq|owl

-m, --max-rows <max_rows>
The maximum number of rows to return

-o, --output <output>

Arguments

FILEPATH
Required argument

transform

Transform a Knowledge Graph from one serialization form to another.

```
KGX CLI transform [OPTIONS] INPUTS...
```

Options

--input-type <input_type>

> **Options** tar|txt|csv|tsv|graphml|ttl|json|rq|owl

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

--output-type <output_type>
[required]

> **Options** tar|txt|csv|tsv|graphml|ttl|json|rq|owl

--mapping <mapping>

--preserve

Arguments

INPUTS
Required argument(s)

validate

Run KGX validation on an input file to check for BioLink Model compliance.

```
KGX CLI validate [OPTIONS] PATH
```

Options

-o, --output <output>
The path to a text file to append the output to. [required]

-d, --output-dir <output_dir>
The path to a directory to save a series of text files to.

Arguments

PATH
Required argument

1.3 Examples

TODO

1.4 KGX CLI usage

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k

INDEX

Symbols

A

E

F

G

I

J

K

L

M

N

O

P

Q

R

S