N2G

Release 0.0.1

Oct 07, 2020

Contents

CHAPTER 1

Installation

Install from PYPI using pip:

```
pip install N2G
```

Or copy repository from GitHub and run:

```
python -m pip install .
```

or:

```
python setup.py install
```

1.1 Additional dependencies

N2G uses mainly Python built-in libraries, but:

- For layout method need Python igraph library to be installed on the system

CHAPTER 2

Overview

N2G is a library to produce XML text files structured in a format supported for opening and editing by these applications:

- yWorsk yEd Graph Editor and yEd web application
- Diagrams DrawIO desktop and DrawIO web application

N2G contains dedicated modules for each format with very similar API that can help create, load, modify and save diagrams.

However, due to discrepancy in functionality and peculiarities of applications itself, N2G modules API is not 100% identical and differ to reflect particular application capabilities.

CHAPTER 3

yEd Module

N2G yEd Module supports producing graphml XML structured text files that can be opened by yWorsk yEd Graph Editor or yEd web application.

3.1 Quick start

Nodes and links can be added one by one using `add_node` and `add_link` methods

```
from N2G import yed_diagram

diagram = yed_diagram()
diagram.add_node('R1', top_label='Core', bottom_label='ASR1004')
diagram.add_node('R2', top_label='Edge', bottom_label='MX240')
diagram.add_link('R1', 'R2', label='DF', src_label='Gi0/1', trgt_label='ge-0/1/2')
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

After opening and editing diagram, it might look like this:

3.2 Adding SVG nodes

By default N2G uses shape nodes, but svg image can be sourced from directory on your system and used as node image instead. However, svg images as nodes can support only one label attribute, that label will be displayed above svg picture.

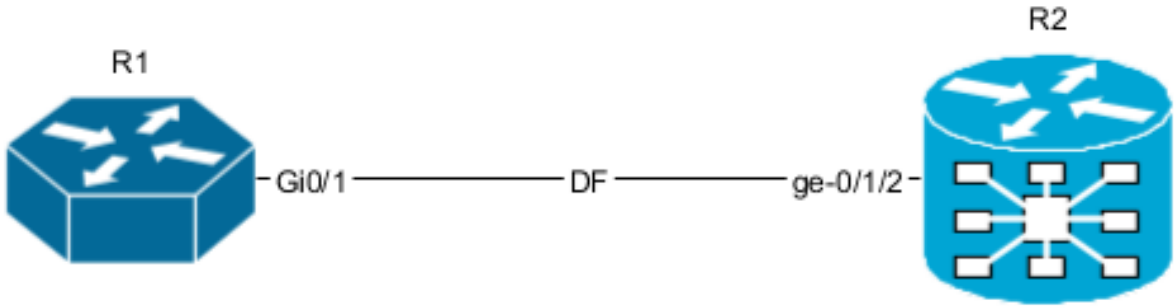
```
from N2G import yed_diagram

diagram = yed_diagram()
diagram.add_node('R1', pic="router.svg", pic_path="./Pics/")
diagram.add_node('R2', pic="router_edge.svg", pic_path="./Pics/")
diagram.add_link('R1', 'R2', label='DF', src_label='Gi0/1', trgt_label='ge-0/1/2')
```

(continues on next page)

(continued from previous page)

```
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

After opening and editing diagram, it might look like this:



3.3 Nodes and links data attributes

Description and URL attributes can be added to node and link. Description attribute can be used by yEd to search for elements as well as diagrams exported in svg format can display data attributes as a tooltips.

```
from N2G import yed_diagram

diagram = yed_diagram()
diagram.add_node('R1', top_label='Core', bottom_label='ASR1004', description=
↪"loopback0: 192.168.1.1", url="google.com")
diagram.add_node('R2', top_label='Edge', bottom_label='MX240', description=
↪"loopback0: 192.168.1.2")
diagram.add_link('R1', 'R2', label='DF', src_label='Gi0/1', trgt_label='ge-0/1/2',␣
↪description="link media-type: 10G-LR", url="github.com")
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

After opening and editing diagram, it might look like this:

Node R1 and link should be clickable on above image as they contain URL information, tooltip should be displayed if svg will be open on its own.

3.4 Loading graph from dictionary

Diagram elements can be loaded from dictionary structure. That dictionary may contain nodes, links and edges keys, these keys should contain list of dictionaries where each dictionary item will contain elements attributes such as id, labels, description etc.

```
from N2G import yed_diagram

diagram = yed_diagram()
sample_graph={
'nodes': [
```

(continues on next page)

(continued from previous page)

```
    {'id': 'a', 'pic': 'router.svg', 'label': 'R1' },
    {'id': 'R2', 'bottom_label':'CE12800', 'top_label':'1.1.1.1'},
    {'id': 'c', 'label': 'R3', 'bottom_label':'FI', 'top_label':'fns751', 'description
↪': 'role: access'},
    {'id': 'd', 'pic':'firewall.svg', 'label': 'FW1', 'description': 'location: US'},
    {'id': 'R4', 'pic': 'router'}
],
'links': [
    {'source': 'a', 'src_label': 'Gig0/0\nUP', 'label': 'DF', 'target': 'R2', 'trgt_
↪label': 'Gig0/1', 'description': 'role: uplink'},
    {'source': 'R2', 'src_label': 'Gig0/0', 'label': 'Copper', 'target': 'c', 'trgt_
↪label': 'Gig0/2'},
    {'source': 'c', 'src_label': 'Gig0/0', 'label': 'ZR', 'target': 'a', 'trgt_label
↪': 'Gig0/3'},
    {'source': 'd', 'src_label': 'Gig0/10', 'label': 'LR', 'target': 'c', 'trgt_label
↪': 'Gig0/8'},
    {'source': 'd', 'src_label': 'Gig0/11', 'target': 'R4', 'trgt_label': 'Gig0/18'}
]}
diagram.from_dict(sample_graph)
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

After opening and editing diagram, it might look like this:

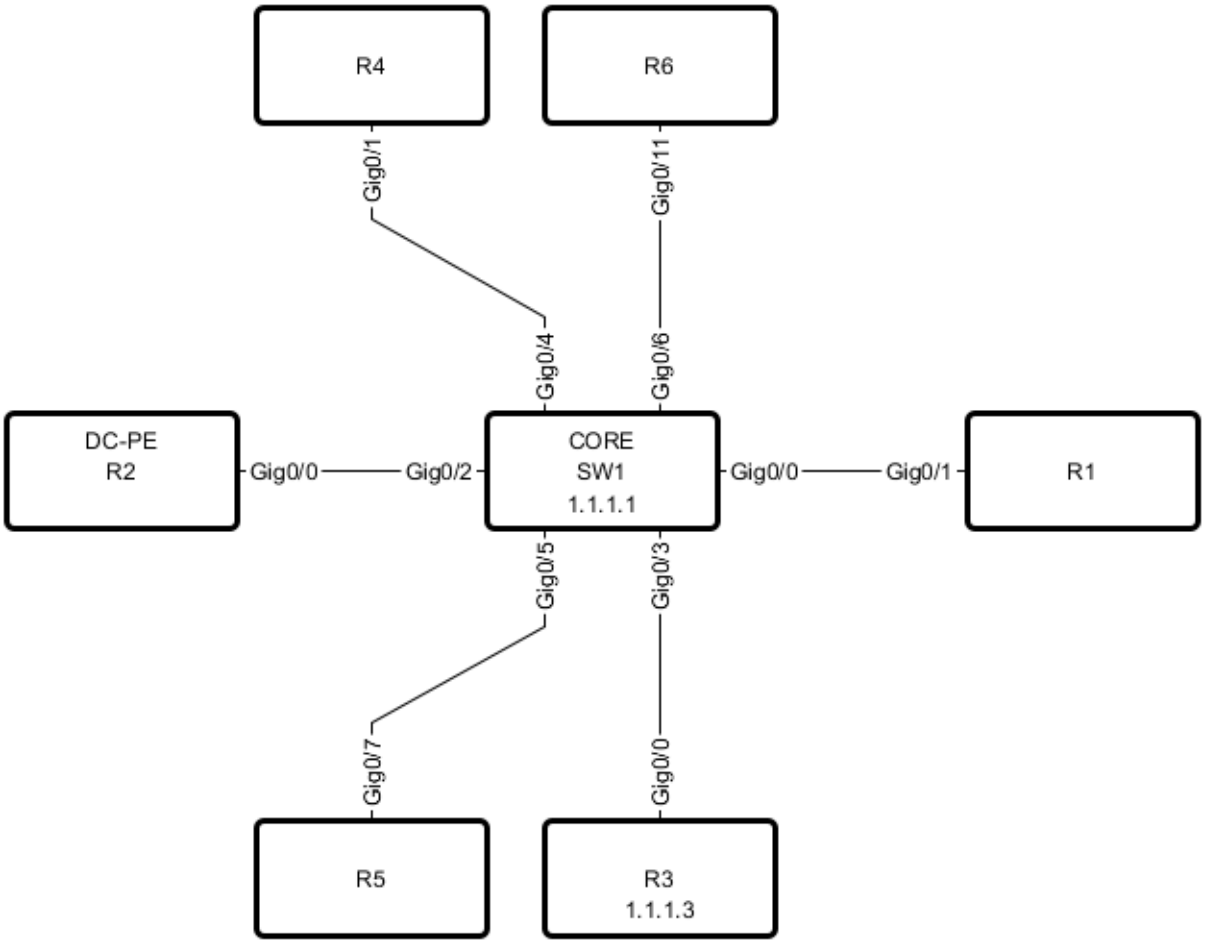
3.5 Loading graph from list

From list method allows to load graph from list of dictionaries, generally containing link details like source, target, labels. Additionally source and target can be defined using dictionaries as well, containing nodes details.

Note: Non-existing node will be automatically added on first encounter, by default later occurrences of same node will not lead to node attributes change, that behavior can be changed setting `node_duplicates` yed_diagram attribute equal to *update* value.

```
from N2G import yed_diagram

diagram = yed_diagram()
sample_list_graph = [
    {'source': {'id': 'SW1', 'top_label': 'CORE', 'bottom_label': '1,1,1,1'}, 'src_
↪label': 'Gig0/0', 'target': 'R1', 'trgt_label': 'Gig0/1'},
    {'source': {'id': 'R2', 'top_label': 'DC-PE'}, 'src_label': 'Gig0/0', 'target':
↪'SW1', 'trgt_label': 'Gig0/2'},
    {'source': {'id':'R3', 'bottom_label': '1.1.1.3'}, 'src_label': 'Gig0/0', 'target
↪': 'SW1', 'trgt_label': 'Gig0/3'},
    {'source': 'SW1', 'src_label': 'Gig0/4', 'target': 'R4', 'trgt_label': 'Gig0/1'},
    {'source': 'SW1', 'src_label': 'Gig0/5', 'target': 'R5', 'trgt_label': 'Gig0/7'},
    {'source': 'SW1', 'src_label': 'Gig0/6', 'target': 'R6', 'trgt_label': 'Gig0/11'}
]
diagram.from_list(sample_list_graph)
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

After opening and editing diagram, it might look like this:



3.6 Loading graph from csv

Similar to `from_dict` or `from_list` methods, from_csv method can take csv data with elements details and add them to diagram. Two types of csv table should be provided - one for nodes, another for links.

```
from N2G import yed_diagram

diagram = yed_diagram()
csv_links_data = """source","src_label","label","target","trgt_label","description"
"a","Gig0/0\nUP","DF","R1","Gig0/1","vlans_trunked: 1,2,3\nstate: up"
"R1","Gig0/0","Copper","c","Gig0/2",
"R1","Gig0/0","Copper","e","Gig0/2",
d,Gig0/21,FW,e,Gig0/23,
"""
csv_nodes_data="""id","pic","label","bottom_label","top_label","description"
a,router,"R12",,,
"R1",,,"SGD1378","servers",
"c",,"R3","SGE3412","servers","1.1.1.1"
"d","firewall.svg","FW1",,,"2.2.2.2"
"e","router","R11",,,
"""
```

(continues on next page)

(continued from previous page)

```
diagram.from_csv(csv_nodes_data)
diagram.from_csv(csv_links_data)
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

After opening and editing diagram, it might look like this:

3.7 Loading existing diagrams

N2G yEd module uses custom `nmetadata` and `emetadata` attributes to store original node and link id. For nodes, `nmetadata` contains node id in a format `{'id': 'node_id_value'}`, for links `emetadata` contains source and target node ids as well as link id, e.g. `{"sid": "SW1", "tid": "R6", "id": "8e96ade0d90d33c3308721dc2a53b391"}`, where link id calculated using rules described in *API reference* section.

`nmetadata` and `emetadata` custom attributes used to properly load previously produced diagrams for modification, as a result:

> **Warning:** currently, N2G yEd module can properly load only diagrams that were created by this module in the first place or diagrams that had manually added `nmetadata` and `emetadata` attributes.

N2G yEd module provides `from_file` and `from_text` methods to load existing diagram content, to load diagram from file one can use this as example:

```
from N2G import yed_diagram

diagram = yed_diagram()
diagram.from_file("./source/diagram_old.graphml")
```

After diagram loaded it can be modified or updated using `add_x`, `from_x`, `delete_x` or `compare` methods.

3.8 Diagram layout

To arrange diagram nodes in certain way one can use `layout` method that relies on igraph library to calculate node coordinates in accordance with certain algorithm. List of supported layout algorithms and their details can be found here together with brief description in *API Reference* section.

Sample code to layout diagram:

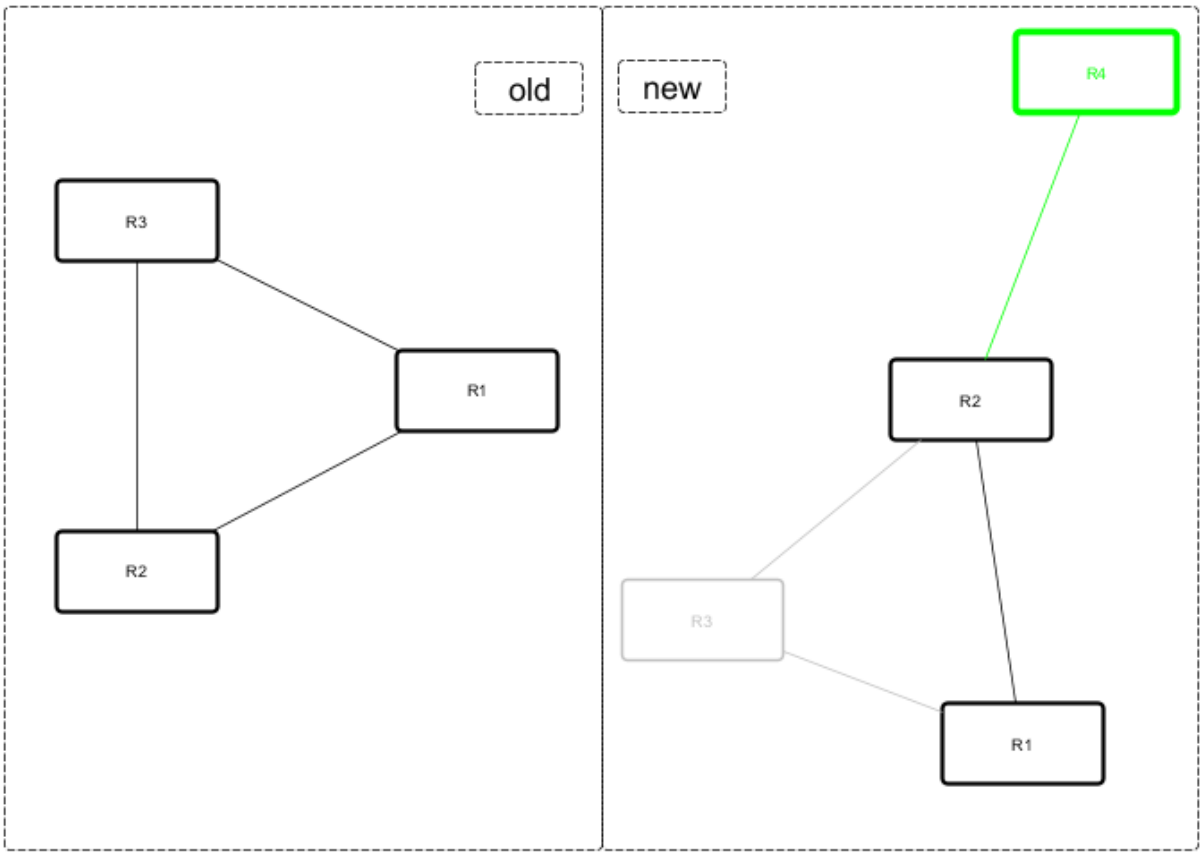
```
from N2G import yed_diagram

diagram = yed_diagram()
diagram.from_file("./source/diagram_old.graphml")
diagram.layout(algo="drl", width=500, height=500)
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

3.9 Comparing diagrams

Comparing diagrams can be useful to spot changes in your system. N2G `compare` method allow to calculate differences between old and new graphs and produce resulting diagram highlighting changes.

```
from N2G import yed_diagram

diagram = yed_diagram()
old_graph = {
'nodes': [
    {'id': 'R1'}, {'id': 'R2'}, {'id': 'R3'},
    ],
'edges': [
    {'source': 'R1', 'target': 'R2'},
    {'source': 'R2', 'target': 'R3'},
    {'source': 'R3', 'target': 'R1'}
]}
new_graph = {
'nodes': [
    {'id': 'R1'}, {'id': 'R2'}, {'id': 'R4'},
    ],
'edges': [
    {'source': 'R1', 'target': 'R2'},
    {'source': 'R2', 'target': 'R4'}
]}
diagram.from_dict(old_graph)
diagram.compare(new_graph)
diagram.layout(algo="kk", width=500, height=500)
diagram.dump_file(filename="Sample_graph.graphml", folder="./Output/")
```

Original and after diagrams:



R3 and its links are missing - highlighted in gray, but R4 and its link is new - highlighted in green.

3.10 API reference

API reference for N2G yEd module.

class N2G.**yed_diagram**(*node_duplicates='skip'*, *link_duplicates='skip'*)

N2G yEd module allows to produce diagrams in yEd .graphml format.

Parameters

- `node_duplicates` (str) can be of value skip, log, update
- `link_duplicates` (str) can be of value skip, log, update

add_link(*source*, *target*, *label=''*, *src_label=''*, *trgt_label=''*, *description=''*, *attributes={}*, *url=''*)

Method to add link between nodes.

Parameters

- `source` (str) mandatory, id of source node
- `target` (str) mandatory, id of target node
- `label` (str) label at the center of the edge, by default equal to id attribute
- `src_label` (str) label to display at the source end of the edge
- `trgt_label` (str) label to display at target end of the edge
- `description` (str) string to save as link `description` attribute
- `url` (str) string to save as link `url` attribute
- `attributes` (dict) dictionary of yEd graphml tag names and attributes

Attributes dictionary keys will be used as xml tag names and values dictionary will be used as xml tag attributes, example:

```
{
    "LineStyle": {"color": "#00FF00", "width": "1.0"},
    "EdgeLabel": {"textColor": "#00FF00"},
}
```

Note: If source or target nodes does not exists, they will be automatically created

add_node(*id*, ***kwargs*)

Convenience method to add node, by calling one of node add methods following these rules:

- If `pic` attribute in kwargs, `add_svg_node` is called
- If `group` kwargs attribute equal to *True*, `_add_group_node` called
- `add_shape_node` called otherwise

Parameters

- `id` (str) mandatory, unique node identifier, usually equal to node name

add_shape_node(*id*, *label=''*, *top_label=''*, *bottom_label=''*, *attributes={}*, *description=''*, *shape_type='roundrectangle'*, *url=''*, *width=120*, *height=60*, *x_pos=200*, *y_pos=150*, ***kwargs*)

Method to add node of type "shape".

Parameters

- `id` (str) mandatory, unique node identifier, usually equal to node name
- `label` (str) label at the center of the node, by default equal to id attribute
- `top_label` (str) label displayed at the top of the node
- `bottom_label` (str) label displayed at the bottom of the node
- `description` (str) string to save as node `description` attribute
- `shape_type` (str) shape type, default - "roundrectangle"
- `url` (str) url string to save a node `url` attribute
- `width` (int) node width in pixels
- `height` (int) node height in pixels
- `x_pos` (int) node position on x axis
- `y_pos` (int) node position on y axis
- `attributes` (dict) dictionary of yEd graphml tag names and attributes

Attributes dictionary keys will be used as xml tag names and values dictionary will be used as xml tag attributes, example:

```
{
    'Shape'     : {'type': 'roundrectangle'},
    'DropShadow': { 'color': '#B3A691', 'offsetX': '5', 'offsetY': '5'}
}
```

add_svg_node(*pic*, *id*, *pic_path='./Pics/'*, *label=''*, *attributes={}*, *description=''*, *url=''*, *width=50*, *height=50*, *x_pos=200*, *y_pos=150*, ***kwargs*)

Method to add SVG picture as node by loading SVG file content into graphml

Parameters

- `id` (str) mandatory, unique node identifier, usually equal to node name
- `pic` (str) mandatory, name of svg file
- `pic_path` (str) OS path to SVG file folder, default is `./Pics/`
- `label` (str) label displayed above SVG node, if not provided, label set equal to id
- `description` (str) string to save as node `description` attribute
- `url` (str) url string to save as node `url` attribute
- `width` (int) node width in pixels
- `height` (int) node height in pixels
- `x_pos` (int) node position on x axis
- `y_pos` (int) node position on y axis
- `attributes` (dict) dictionary of yEd graphml tag names and attributes

Attributes dictionary keys will be used as xml tag names and values dictionary will be used as xml tag attributes, example:

```
{
    'DropShadow': { 'color': '#B3A691', 'offsetX': '5', 'offsetY': '5'}
}
```

compare(*data*, *missing_nodes={'BorderStyle': {'color': '#C0C0C0', 'width': '2.0'}, 'NodeLabel': {'textColor': '#C0C0C0'}}*, *new_nodes={'BorderStyle': {'color': '#00FF00', 'width': '5.0'}, 'NodeLabel': {'textColor': '#00FF00'}}*, *missing_links={'EdgeLabel': {'textColor': '#C0C0C0'}, 'LineStyle': {'color': '#C0C0C0', 'width': '1.0'}}*, *new_links={'EdgeLabel': {'textColor': '#00FF00'}, 'LineStyle': {'color': '#00FF00', 'width': '1.0'}}*)

Method to combine two graphs - existing and new - and produce resulting graph following these rules:

- nodes and links present in new graph but not in existing graph considered as new and will be updated with `new_nodes` and `new_links` attributes by default highlighting them in green
- nodes and links missing from new graph but present in existing graph considered as missing and will be updated with `missing_nodes` and `missing_links` attributes by default highlighting them in gray
- nodes and links present in both graphs will remain unchanged

Parameters

- `data` (dict) dictionary containing new graph data, dictionary format should be the same as for `from_dict` method.
- `missing_nodes` (dict) dictionary with attributes to apply to missing nodes
- `new_nodes` (dict) dictionary with attributes to apply to new nodes
- `missing_links` (dict) dictionary with attributes to apply to missing links
- `new_links` (dict) dictionary with attributes to apply to new links

Sample usage:

```
from N2G import yed_diagram
diagram = yed_diagram()
new_graph = {
    'nodes': [
        {'id': 'a', 'pic': 'router_round', 'label': 'R1' }
    ],
    'edges': [
        {'source': 'f', 'src_label': 'Gig0/21', 'label': 'DF', 'target': 'b'}
    ]
}
diagram.from_file("./old_graph.graphml")
diagram.compare(new_graph)
diagram.dump_file(filename="compared_graph.graphml")
```

delete_link(*id=None*, *ids=[]*, *label=''*, *src_label=''*, *trgt_label=''*, *source=''*, *target=''*)

Method to delete link by its id. Bulk delete operation supported by providing list of link ids to delete.

If link `id` or `ids` not provided, id calculated based on - `label, src_label, trgt_label, source, target` - attributes using this algorithm:

1. Edge tuple produced: `tuple(sorted([label, src_label, trgt_label, source, target]))`

2. MD5 hash derived from tuple: `hashlib.md5(",".join(edge_tup).encode()).hexdigest()`

Parameters

- `id` (str) id of single link to delete
- `ids` (list) list of link ids to delete
- `label` (str) link label to calculate id of single link to delete
- `src_label` (str) link source label to calculate id of single link to delete
- `trgt_label` (str) link target label to calculate id of single link to delete
- `source` (str) link source to calculate id of single link to delete
- `target` (str) link target to calculate id of single link to delete

delete_node(*id=None*, *ids=[]*)

Method to delete node by its id. Bulk delete operation supported by providing list of node ids to delete.

Parameters

- `id` (str) id of single node to delete
- `ids` (list) list of node ids to delete

dump_file(*filename=None*, *folder='./Output/'*)

Method to save current diagram in .graphml file.

Parameters

- `filename` (str) name of the file to save diagram into
- `folder` (str) OS path to folder where to save diagram file

If no `filename` provided, timestamped format will be used to produce filename, e.g.: `Sun Jun 28 20-30-57 2020_output.graphml`

dump_xml()

Method to return current diagram XML text

from_csv(*text_data*)

Method to build graph from CSV tables

Parameters

- `text_data` (str) CSV text with links or nodes details

This method supports loading CSV text data that contains nodes or links information. If `id` in headers, `from_dict` method will be called for CSV processing, `from_list` method will be used otherwise.

CSV data with nodes details should have headers matching add node methods arguments and rules.

CSV data with links details should have headers matching `add_link` method arguments and rules.

Sample CSV table with link details:

```
"source","src_label","label","target","trgt_label","description"
"a","Gig0/0","DF","b","Gig0/1","vlans_trunked: 1,2,3"
"b","Gig0/0","Copper","c","Gig0/2",
"b","Gig0/0","Copper","e","Gig0/2",
d,Gig0/21,FW,e,Gig0/23,
```

Sample CSV table with node details:

```
"id","pic","label","bottom_label","top_label","description"
a,router_1,"R1,2",,,
"b",,,"some","top_some",
"c",,"somelabel","botlabel","toplabel","some node description"
"d","firewall.svg","somelabel1",,,"some node description"
"e","router_2","R1",,,
```

from_dict(*data*)

Method to build graph from dictionary.

Parameters

- `data` (dict) dictionary with nodes and link/edges details.

Example `data` dictionary:

```
sample_graph = {
    'nodes': [
        {
            'id': 'a',
            'pic': 'router',
            'label': 'R1'
        },
        {
            'id': 'b',
            'label': 'somelabel',
            'bottom_label':'botlabel',
            'top_label':'toplabel',
            'description': 'some node description'
        },
        {
            'id': 'e',
            'label': 'E'
        }
    ],
    'edges': [
        {
            'source': 'a',
            'src_label': 'Gig0/0',
            'label': 'DF',
            'target': 'b',
            'trgt_label': 'Gig0/1',
            'description': 'vlans_trunked: 1,2,3'
        }
    ],
    'links': [
        {
            'source': 'a',
            'target': 'e'
        }
    ]
}
```

Dictionary Content Rules

- dictionary may contain `nodes` key with a list of nodes dictionaries
- each node dictionary must contain unique `id` attribute, other attributes are optional
- dictionary may contain `edges` or `links` key with a list of edges dictionaries

- each link dictionary must contain `source` and `target` attributes, other attributes are optional

from_file (*filename*, *file_load='xml'*)
Method to load data from file for processing. File format can be yEd graphml (XML) or CSV

Parameters

- `filename` (str) OS path to file to load
- `file_load` (str) indicated the load of the file, supports `xml`, `csv`

from_list (*data*)
Method to build graph from list.

Parameters

- `data` (list) list of link dictionaries,

Example `data` list:

```
sample_graph = [
    {
        'source': 'a',
        'src_label': 'Gig0/0\nUP',
        'label': 'DF',
        'target': 'b',
        'trgt_label': 'Gig0/1',
        'description': 'vlans_trunked: 1,2,3\nstate: up'
    },
    {
        'source': 'a',
        'target': {
                'id': 'e',
                'label': 'somelabel',
                'bottom_label':'botlabel',
                'top_label':'toplabel',
                'description': 'some node description'
            }
        }
    }
]
```

List Content Rules

- each list item must have `target` and `source` attributes defined
- `target`/`source` attributes can be either a string or a dictionary
- dictionary `target`/`source` node must contain `id` attribute and other supported node attributes

---

Note: By default yed_diagram object `node_duplicates` action set to 'skip' meaning that node will be added on first occurrence and ignored after that. Set `node_duplicates` to 'update' if node with given id need to be updated by later occurrences in the list.

---

from_xml (*text_data*)
Method to load yEd graphml XML formatted text for processing

Parameters

- `text_data` (str) text data to load

layout (*algo='kk'*, *width=1360*, *height=864*, ***kwargs*)
Method to calculate graph layout using Python igraph library

Parameters

- `algo` (str) name of layout algorithm to use, default is 'kk'. Reference *Layout algorithms* table below for valid algo names
- `width` (int) width in pixels to fit layout in
- `height` (int) height in pixels to fit layout in
- `kwargs` any additional kwargs to pass to igraph `Graph.layout` method

Layout algorithms

algo name	description
circle, circular	Deterministic layout that places the vertices on a circle
drl	The Distributed Recursive Layout algorithm for large graphs
fr	Fruchterman-Reingold force-directed algorithm
fr3d, fr_3d	Fruchterman-Reingold force-directed algorithm in three dimensions
grid_fr	Fruchterman-Reingold force-directed algorithm with grid heuristics for large graphs
kk	Kamada-Kawai force-directed algorithm
kk3d, kk_3d	Kamada-Kawai force-directed algorithm in three dimensions
large, lgl, large_graph	The Large Graph Layout algorithm for large graphs
random	Places the vertices completely randomly
random_3d	Places the vertices completely randomly in 3D
rt, tree	Reingold-Tilford tree layout, useful for (almost) tree-like graphs
rt_circular, tree	Reingold-Tilford tree layout with a polar coordinate post-transformation, useful for (almost) tree-like graphs
sphere, spherical, circular_3d	Deterministic layout that places the vertices evenly on the surface of a sphere

set_attributes (*element*, *attributes={}*)
Method to set attributes for XML element

Parameters

- `element` (object) xml etree element object to set attributes for
- `attributes` (dict) dictionary of yEd graphml tag names and attributes

Attributes dictionary keys will be used as xml tag names and values dictionary will be used as xml tag attributes, example:

```
{
    "LineStyle": {"color": "#00FF00", "width": "1.0"},
    "EdgeLabel": {"textColor": "#00FF00"},
}
```

update_link (*edge_id=''*, *label=''*, *src_label=''*, *trgt_label=''*, *source=''*, *target=''*, *new_label=None*, *new_src_label=None*, *new_trgt_label=None*, *description=''*, *attributes={}*)
Method to update edge/link details.

Parameters

- `edge_id` (str) md5 hash edge id, if not provided, will be generated based on edge attributes

- `label` (str) existing edge label
- `src_label` (str) existing edge src_label
- `trgt_label` (str) existing edge tgt_label
- `source` (str) existing edge source node ID
- `target` (str) existing edge target node id
- `new_label` (str) new edge label
- `new_src_label` (str) new edge src_label
- `new_trgt_label` (str) new edge tgt_label
- `description` (str) new edge description
- `attributes` (str) dictionary of attributes to apply to edge element

Either of these must be provided to find edge element to update:

- `edge_id` MD5 hash or
- `label, src_label, trgt_label, source, target` attributes to calculate `edge_id`

`edge_id` calculated based on - `label, src_label, trgt_label, source, target` - attributes following this algorithm:

1. Edge tuple produced: `tuple(sorted([label, src_label, trgt_label, source, target]))`
2. MD5 hash derived from tuple: `hashlib.md5(",".join(edge_tup).encode()).hexdigest()`

This method will replace existing and add new labels to the link.

Existing description attribute will be replaced with new value.

Attributes will replace existing values.

update_node(*id*, *label=None*, *top_label=None*, *bottom_label=None*, *attributes={}*, *description=None*, *width=''*, *height=''*)

Method to update node details

Parameters

- `id` (str) mandatory, unique node identifier, usually equal to node name
- `label` (str) label at the center of the shape node or above SVG node
- `top_label` (str) label displayed at the top of the node
- `bottom_label` (str) label displayed at the bottom of the node
- `description` (str) string to save as node `description` attribute
- `width` (int) node width in pixels
- `height` (int) node height in pixels
- `attributes` (dict) dictionary of yEd graphml tag names and attributes

Attributes dictionary keys will be used as xml tag names and values dictionary will be used as xml tag attributes, example:

```
{
    'Shape'      : {'type': 'roundrectangle'},
    'DropShadow': { 'color': '#B3A691', 'offsetX': '5', 'offsetY': '5'}
}
```

This method will replace existing and add new labels to the node.

Existing description attribute will be replaced with new value.

Height and width will override existing values.

Attributes will replace existing values.

CHAPTER 4

Drawlo Module

N2G Drawio Module supports producing XML structured text files that can be opened by Diagrams DrawIO desktop or DrawIO web applications

4.1 Quick start

Nodes and links can be added one by one using `add_node` and `add_link` methods

```
from N2G import drawio_diagram

diagram = drawio_diagram()
diagram.add_diagram("Page-1")
diagram.add_node(id="R1")
diagram.add_node(id="R2")
diagram.add_link("R1", "R2", label="DF")
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.drawio", folder="./Output/")
```

After opening and editing diagram, it might look like this:

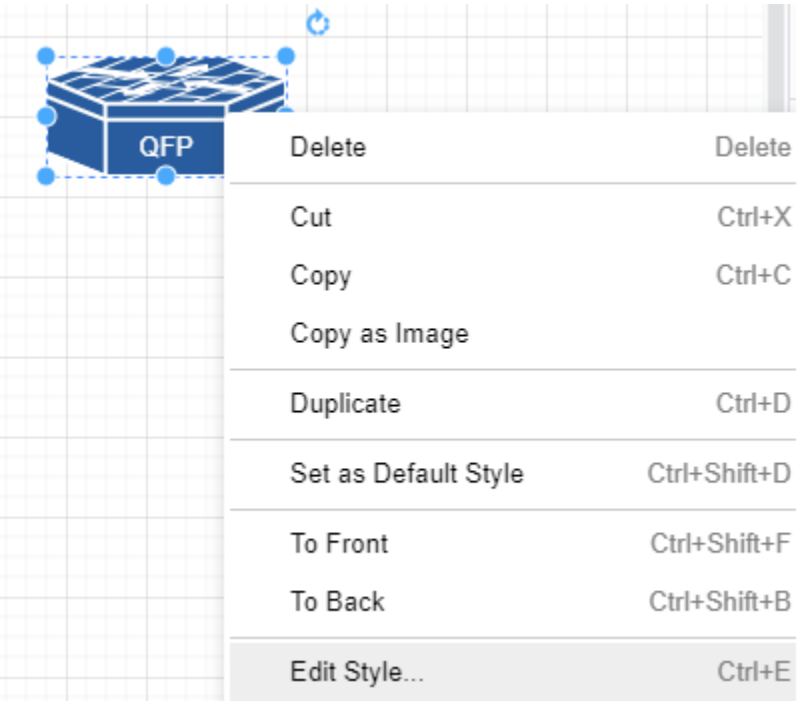
Working with drawio module should be started with adding new diagram, after that nodes and links can be added. It is possible to switch between diagrams to edit using `go_to_diagram` method.

4.2 Adding styles

Styles used to change the way how things look and can be applied to nodes or links. Styles attributes in DrawIO encoded using strings similar to this one:

```
shape=mxgraph.cisco.misc.asr_1000_series;html=1;pointerEvents=1;dashed=0;fillColor=
↪#036897;strokeColor=#ffffff;strokeWidth=2;verticalLabelPosition=bottom;
↪verticalAlign=top;align=center;outlineConnect=0;
```

above strings can be found in node and link settings:



and can be used to reference by node and links style `attribute`, additionally, style string can be saved in text file and `style` attribute can reference that file OS path location.

```
from N2G import drawio_diagram

new_link_style="endArrow=classic;fillColor=#f8cecc;strokeColor=#FF3399;dashed=1;
↪edgeStyle=entityRelationEdgeStyle;startArrow=diamondThin;startFill=1;endFill=0;
↪strokeWidth=5;"
building_style="shape=mxgraph.cisco.buildings.generic_building;html=1;pointerEvents=1;
↪dashed=0;fillColor=#036897;strokeColor=#ffffff;strokeWidth=2;
↪verticalLabelPosition=bottom;verticalAlign=top;align=center;outlineConnect=0;"

diagram = drawio_diagram()
diagram.add_diagram("Page-1")
diagram.add_node(id="HQ", style=building_style, width=90, height=136)
diagram.add_node(id="R1", style="./styles/router.txt")
diagram.add_link("R1", "HQ", label="DF", style=new_link_style)
```

where *./styles/router.txt* content is:

```
shape=mxgraph.cisco.routers.atm_router;html=1;pointerEvents=1;dashed=0;fillColor=
↪#036897;strokeColor=#ffffff;strokeWidth=2;verticalLabelPosition=bottom;
↪verticalAlign=top;align=center;outlineConnect=0;
```

After opening and editing diagram, it might look like this:

Note: DrawIO does not encode node width and height attributes in style string, as a result width and height should be provided separately or will be set to default values: 120 and 60 pixels

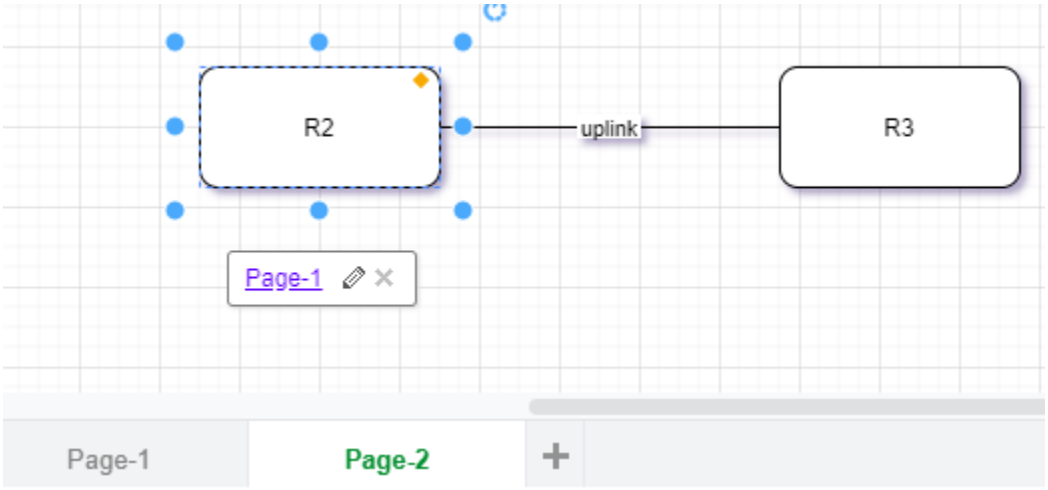
4.3 Nodes and links data attributes

Data and URL attributes can be added to links and nodes to encode additional information. Data attribute should be a dictionary of key value pairs to add, where values can be of type string only.

URL attribute can point to WEB link or can reference another diagram/tab name.

```
from N2G import drawio_diagram

diagram = drawio_diagram()
diagram.add_diagram("Page-1")
diagram.add_node(id="R1", data={"a": "b", "c": "d"}, url="http://google.com")
diagram.add_diagram("Page-2")
diagram.add_node(id="R2", url="Page-1")
diagram.add_node(id="R3")
diagram.add_link("R2", "R3", label="uplink", data={"speed": "1G", "media": "10G-LR"},␣
↪url="http://cmdb.local")
diagram.dump_file(filename="Sample_graph.drawio", folder="./Output/")
```

After opening and editing diagram, it might look like this:



4.4 Loading graph from dictionary

Diagram elements can be loaded from dictionary structure. That dictionary may contain nodes, links and edges keys, these keys should contain list of dictionaries where each dictionary item will contain element attributes such as id, labels, data, url etc.

```
from N2G import drawio_diagram

diagram = drawio_diagram()
sample_graph={
'nodes': [
    {'id': 'a', 'style': './styles/router.txt', 'label': 'R1', 'width': 78, 'height':␣
↪53},
    {'id': 'R2', 'label':'CE12800'},
    {'id': 'c', 'label': 'R3', 'data': {'role': 'access', 'make': 'VendorX'}}
],
'links': [
```

(continues on next page)

(continued from previous page)

```
    {'source': 'a', 'label': 'DF', 'target': 'R2', 'data': {'role': 'uplink'}},
    {'source': 'R2', 'label': 'Copper', 'target': 'c'},
    {'source': 'c', 'label': 'ZR', 'target': 'a'}
]}
diagram.from_dict(sample_graph, width=300, height=200, diagram_name="Page-2")
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.drawio", folder="./Output/")
```

After opening and editing diagram, it might look like this:

4.5 Loading graph from list

From list method allows to load graph from list of dictionaries, generally containing link details like source, target, label. Additionally source and target can be defined using dictionaries as well, containing nodes details.

Note: Non-existing node will be automatically added on first encounter, by default later occurrences of same node will not lead to node attributes change, that behavior can be changed setting `node_duplicates` drawio_diagram attribute equal to *update* value.

```
from N2G import drawio_diagram

diagram = drawio_diagram()
sample_list_graph = [
    {'source': {'id': 'SW1'}, 'target': 'R1', 'label': 'Gig0/1--Gi2'},
    {'source': 'R2', 'target': 'SW1', "data": {"speed": "1G", "media": "10G-LR"}},
    {'source': {'id':'a', 'label': 'R3'}, 'target': 'SW1'},
    {'source': 'SW1', 'target': 'R4'}
]
diagram.from_list(sample_list_graph, width=300, height=200, diagram_name="Page-2")
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.drawio", folder="./Output/")
```

After opening and editing diagram, it might look like this:

4.6 Loading graph from csv

Similar to `from_dict` or `from_list`, from_csv method can take csv data with elements details and add them to diagram. Two types of csv table should be provided - one for nodes, another for links.

```
from N2G import drawio_diagram

diagram = drawio_diagram()
csv_links_data = """source","label","target"
"a","DF","b"
"b","Copper","c"
"b","Copper","e"
d,FW,e
"""
csv_nodes_data="""id","label","style","width","height"
a,"R12","./styles/router.txt",78,53
```

(continues on next page)

(continued from previous page)

```
"b","R2",,,
"c","R3",,,
"d","SW22",,,
"e","R1",,,
"""
diagram.from_csv(csv_nodes_data)
diagram.from_csv(csv_links_data)
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.drawio", folder="./Output/")
```

After opening and editing diagram, it might look like this:

4.7 Loading existing diagrams

N2G DrawIO module provides `from_file` and `from_text` methods to load existing diagram content, to load diagram from file one can use this as example:

```
from N2G import drawio_diagram

diagram = drawio_diagram()
drawing.from_file("./source/old_office_diagram.drawio")
```

After diagram loaded it can be modified or updated using `add_x`, `from_x`, `delete_x` or `compare` methods.

4.8 Diagram layout

To arrange diagram nodes in certain way one can use `layout` method that relies on igraph library to calculate node coordinates in accordance with certain algorithm. List of supported layout algorithms and their details can be found here together with brief description in *API Reference* section.

Sample code to layout diagram:

```
from N2G import drawio_diagram

diagram = drawio_diagram()
diagram.from_file("./source/old_office_diagram.graphml")
diagram.layout(algo="drl")
diagram.dump_file(filename="updated_office_diagram.graphml", folder="./Output/")
```

4.9 Comparing diagrams

Comparing diagrams can help to spot changes in your system. N2G `compare` method allow to calculate differences between old and new graphs and produce resulting diagram highlighting changes.

```
from N2G import drawio_diagram

diagram = drawio_diagram()
old_graph = {
'nodes': [
    {'id': 'R1'}, {'id': 'R2'}, {'id': 'R3'},
```

(continues on next page)

(continued from previous page)

```
    ],
'edges': [
    {'source': 'R1', 'target': 'R2'},
    {'source': 'R2', 'target': 'R3'},
    {'source': 'R3', 'target': 'R1'}
]}
new_graph = {
'nodes': [
    {'id': 'R1'}, {'id': 'R2'}, {'id': 'R4'},
    ],
'edges': [
    {'source': 'R1', 'target': 'R2'},
    {'source': 'R2', 'target': 'R4'}
]}
diagram.add_diagram("Page-1", width=500, height=500)
diagram.from_dict(old_graph)
diagram.compare(new_graph)
diagram.layout(algo="kk")
diagram.dump_file(filename="Sample_graph.drawio", folder="./Output/")
```

Original and after diagrams combined:

R3 and its links are missing - highlighted in gray, but R4 and its link is new - highlighted in green.

4.10 API reference

API reference for N2G DrawIO module.

class N2G.**drawio_diagram**(*node_duplicates='skip'*, *link_duplicates='skip'*)

N2G DrawIO module allows to produce diagrams compatible with DrawIO XML format.

Parameters

- `node_duplicates` (str) can be of value skip, log, update
- `link_duplicates` (str) can be of value skip, log, update

add_diagram(*id*, *name=''*, *width=1360*, *height=864*)

Method to add new diagram tab and switch to it.

> **Warning:** This method must be called to create at list one diagram tab to work with prior to nodes and links can be added to the drawing calling `add_link` or `add_node` methods.

Parameters

- `id` (str) id of the diagram, should be unique across other diagrams
- `name` (str) tab name
- `width` (int) width of diagram in pixels
- `height` (int) height of diagram in pixels

add_link(*source*, *target*, *style=''*, *label=''*, *data={}*, *url=''*)

Method to add link between nodes to the diagram.

Parameters

- `source` (str) mandatory, source node id
- `source` (str) mandatory, target node id
- `label` (str) link label to display at the centre of the link
- `data` (dict) dictionary of key value pairs to add as link data
- `url` (str) url string to save as link `url` attribute
- `style` (str) string containing DrawIO style parameters to apply to the link

Sample DrawIO style string for the link:

```
endArrow=classic;fillColor=#f8cecc;strokeColor=#FF3399;dashed=1;
edgeStyle=entityRelationEdgeStyle;startArrow=diamondThin;startFill=1;
endFill=0;strokeWidth=5;
```

Note: If source or target nodes does not exists, they will be automatically created

add_node(*id*, *label=''*, *data={}*, *url=''*, *style=''*, *width=120*, *height=60*, *x_pos=200*, *y_pos=150*)
Method to add node to the diagram.

Parameters

- `id` (str) mandatory, unique node identifier, usually equal to node name
- `label` (str) node label, if not provided, set equal to id
- `data` (dict) dictionary of key value pairs to add as node data
- `url` (str) url string to save as node `url` attribute
- `width` (int) node width in pixels
- `height` (int) node height in pixels
- `x_pos` (int) node position on x axis
- `y_pos` (int) node position on y axis
- `style` (str) string containing DrawIO style parameters to apply to the node

Sample DrawIO style string for the node:

```
shape=mxgraph.cisco.misc.asr_1000_series;html=1;pointerEvents=1;
dashed=0;fillColor=#036897;strokeColor=#ffffff;strokeWidth=2;
verticalLabelPosition=bottom;verticalAlign=top;align=center;
outlineConnect=0;
```

compare(*data*, *diagram_name=None*, *missing_colour='#C0C0C0'*, *new_colour='#00FF00'*)
Method to combine two graphs - existing and new - and produce resulting graph following these rules:

- nodes and links present in new graph but not in existing graph considered as new and will be updated with `new_colour` style attribute by default highlighting them in green
- nodes and links missing from new graph but present in existing graph considered as missing and will be updated with `missing_colour` style attribute by default highlighting them in gray
- nodes and links present in both graphs will remain unchanged

Parameters

- `data` (dict) dictionary containing new graph data, dictionary format should be the same as for `from_dict` method.
- `missing_colour` (str) colour to apply to missing elements
- `new_colour` (str) colour to apply to new elements

Sample usage:

```
from N2G import drawio_diagram
existing_graph = {
    "nodes": [
        {"id": "node-1"},
        {"id": "node-2"},
        {"id": "node-3"}
],
    "links": [
        {"source": "node-1", "target": "node-2", "label": "bla1"},
        {"source": "node-2", "target": "node-3", "label": "bla2"},
    ]
}
new_graph = {
    "nodes": [
        {"id": "node-99"},
        {"id": "node-100", "style": "./Pics/router_1.txt", "width": 78,
↪"height": 53},
    ],
    "links": [
        {"source": "node-2", "target": "node-3", "label": "bla2"},
        {"source": "node-99", "target": "node-3", "label": "bla99"},
        {"source": "node-100", "target": "node-99", "label": "bla10099"},
    ]
}
drawing = drawio_diagram()
drawing.from_dict(data=existing_graph)
drawing.compare(new_graph)
drawing.layout(algo="kk")
drawing.dump_file(filename="compared_graph.drawio")
```

delete_link(*id=None*, *ids=[]*, *label=''*, *source=''*, *target=''*)

Method to delete link by its id. Bulk delete operation supported by providing list of link ids to delete.

If link `id` or `ids` not provided, id calculated based on - `label, source, target` - attributes using this algorithm:

1. Edge tuple produced: `tuple(sorted([label, source, target]))`
2. MD5 hash derived from tuple: `hashlib.md5(",".join(edge_tup).encode()).hexdigest()`

Parameters

- `id` (str) id of single link to delete
- `ids` (list) list of link ids to delete
- `label` (str) link label to calculate id of single link to delete
- `source` (str) link source to calculate id of single link to delete
- `target` (str) link target to calculate id of single link to delete

delete_node(*id=None*, *ids=[]*)
Method to delete node by its id. Bulk delete operation supported by providing list of node ids to delete.

Parameters

- `id` (str) id of single node to delete
- `ids` (list) list of node ids to delete

dump_file(*filename=None*, *folder='./Output/'*)
Method to save current diagram in .drawio file.

Parameters

- `filename` (str) name of the file to save diagram into
- `folder` (str) OS path to folder where to save diagram file

If no `filename` provided, timestamped format will be used to produce filename, e.g.: `Sun Jun 28 20-30-57 2020_output.drawio`

dump_xml()
Method to return current diagram XML text

from_csv(*text_data*)
Method to build graph from CSV tables

Parameters

- `text_data` (str) CSV text with links or nodes details

This method supports loading CSV text data that contains nodes or links information. If `id` in headers, `from_dict` method will be called for CSV processing, `from_list` method will be used otherwise.

CSV data with nodes details should have headers matching `add_node` method arguments and rules.

CSV data with links details should have headers matching `add_link` method arguments and rules.

Sample CSV table with links details:

```
"source","label","target"
"a","DF","b"
"b","Copper","c"
"b","Copper","e"
"d","FW","e"
```

Sample CSV table with nodes details:

```
"id","label","style","width","height"
a,"R1,2","./Pics/cisco_router.txt",78,53
"b","some",,,
"c","somelabel",,,
"d","somelabel1",,,
"e","R1",,,
```

from_dict(*data*, *diagram_name='Page-1'*, *width=1360*, *height=864*)
Method to build graph from dictionary.

Parameters

- `diagram_name` (str) name of the diagram tab where to add links and nodes. Diagram tab will be created if it does not exists
- `width` (int) diagram width in pixels

- `height` (int) diagram height in pixels
- `data` (dict) dictionary with nodes and link/edges details, example:

```
sample_graph = {
    'nodes': [
        {
            'id': 'a',
            'label': 'R1'
        },
        {
            'id': 'b',
            'label': 'somelabel',
            'data': {'description': 'some node description'}
        },
        {
            'id': 'e',
            'label': 'E'
        }
    ],
    'edges': [
        {
            'source': 'a',
            'label': 'DF',
            'target': 'b',
            'url': 'google.com'
        }
    ],
    'links': [
        {
            'source': 'a',
            'target': 'e'
        }
    ]
}
```

Dictionary Content Rules

- dictionary may contain `nodes` key with a list of nodes dictionaries
- each node dictionary must contain unique `id` attribute, other attributes are optional
- dictionary may contain `edges` or `links` key with a list of edges dictionaries
- each link dictionary must contain `source` and `target` attributes, other attributes are optional

from_file (*filename*, *file_load='xml'*)

Method to load nodes and links from Drawio diagram file for further processing

Args

- filename - OS path to .drawio file to load

from_list (*data*, *diagram_name='Page-1'*, *width=1360*, *height=864*)

Method to build graph from list.

Parameters

- `diagram_name` (str) name of the diagram tab where to add links and nodes. Diagram tab will be created if it does not exists
- `width` (int) diagram width in pixels

- `height` (int) diagram height in pixels
- `data` (list) list of link dictionaries, example:

```
sample_graph = [
    {
        'source': 'a',
        'label': 'DF',
        'target': 'b',
        'data': {'vlans': 'vlans_trunked: 1,2,3\nstate: up'}
    },
    {
        'source': 'a',
        'target': {
                'id': 'e',
                'label': 'somelabel',
                'data': {'description': 'some node description'}
            }
        }
    }
]
```

List Content Rules

- each list item must have `target` and `source` attributes defined
- `target`/`source` attributes can be either a string or a dictionary
- dictionary `target`/`source` node must contain `id` attribute and other supported node attributes

---

Note: By default drawio_diagram object `node_duplicates` action set to 'skip' meaning that node will be added on first occurrence and ignored after that. Set `node_duplicates` to 'update' if node with given id need to be updated by later occurrences in the list.

`from_xml`(*text_data*)
Method to load graph from .drawio XML text produced by DrawIO

Args

- text_data - text data to load

`go_to_diagram`(*diagram_name=None*, *diagram_index=None*)
DrawIO supports adding multiple diagram tabs within single document. This method allows to switch between diarams in different tabs. That way each tab can be updated separately.

Parameters

- `diagram_name` (str) name of diagram tab to switch to
- `diagram_index` (int) index of diagram tab to switch to, will change to last tab if index is out of range. Index can be positive or negative number and follows Python list index behaviour. For instance, index equal to "-1" we go to last tab, "0" will go to first tab

`layout`(*algo='kk'*, ***kwargs*)
Method to calculate graph layout using Python igraph library

Parameters

- `algo` (str) name of layout algorithm to use, default is 'kk'. Reference *Layout algorithms* table below for valid algo names
- `kwargs` any additional kwargs to pass to igraph `Graph.layout` method

Layout algorithms

algo name	description
circle, circular	Deterministic layout that places the vertices on a circle
drl	The Distributed Recursive Layout algorithm for large graphs
fr	Fruchterman-Reingold force-directed algorithm
fr3d, fr_3d	Fruchterman-Reingold force-directed algorithm in three dimensions
grid_fr	Fruchterman-Reingold force-directed algorithm with grid heuristics for large graphs
kk	Kamada-Kawai force-directed algorithm
kk3d, kk_3d	Kamada-Kawai force-directed algorithm in three dimensions
large, lgl, large_graph	The Large Graph Layout algorithm for large graphs
random	Places the vertices completely randomly
random_3d	Places the vertices completely randomly in 3D
rt, tree	Reingold-Tilford tree layout, useful for (almost) tree-like graphs
rt_circular, tree	Reingold-Tilford tree layout with a polar coordinate post-transformation, useful for (almost) tree-like graphs
sphere, spherical, circular_3d	Deterministic layout that places the vertices evenly on the surface of a sphere

update_link(*edge_id=”*, *label=”*, *source=”*, *target=”*, *new_label=None*, *data={}*, *url=”*, *style=”*)

Method to update edge/link details.

Parameters

- `edge_id` (str) - md5 hash edge id, if not provided, will be generated based on link attributes
- `label` (str) - existing edge label
- `source` (str) - existing edge source node id
- `target` (str) - existing edge target node id
- `new_label` (str) - new edge label
- `data` (str) - edge new data attributes
- `url` (str) - edge new url attribute
- `style` (str) - OS path to file or sting containing edge style

Either of these must be provided to find link element to update:

- `edge_id` MD5 hash or
- `label, source, target` attributes to calculate `edge_id`

`edge_id` calculated based on - `label, source, target` - attributes following this algorithm:

1. Edge tuple produced: `tuple(sorted([label, source, target]))`
2. MD5 hash derived from tuple: `hashlib.md5(",".join(edge_tup).encode()).hexdigest()`

This method will replace existing or add new label to the link.

Existing data attribute will be amended with new values using dictionary like update method.

New style will replace existing style.

update_node(*id*, *label=None*, *data={}*, *url=None*, *style=''*, *width=''*, *height=''*)
Method to update node details. Uses node `id` to search for node to update

Parameters

- `id` (str) mandatory, unique node identifier
- `label` (str) label at the center of the node
- `data` (dict) dictionary of data items to add to the node
- `width` (int) node width in pixels
- `height` (int) node height in pixels
- `url` (str) url string to save as node *url* attribute
- `style` (str) string containing DrawIO style parameters to apply to the node

Python Module Index

n

Index