

---

# **PyGraphistry Documentation**

**Graphistry, Inc.**

**Aug 15, 2021**



CONTENTS

- 1 graphistry package 3
  - 1.1 Submodules 3
  - 1.2 graphistry.plotter module 3
  - 1.3 graphistry.pygraphistry module 3
  - 1.4 graphistry.arrow\_uploader module 40
  - 1.5 graphistry.ArrowFileUploader module 42
- 2 doc 45
  - 2.1 versioneer module 45
- 3 Indices and tables 47
- Index 49



Quickstart: [Read our tutorial](#)



## GRAPHISTORY PACKAGE

### 1.1 Submodules

### 1.2 graphistry.plotter module

```
class graphistry.plotter.Plotter(*args, **kwargs)
    Bases: graphistry.gremlin.CosmosMixin, graphistry.gremlin.NeptuneMixin,
            graphistry.gremlin.GremlinMixin, graphistry.layouts.LayoutsMixin,
            graphistry.compute.ComputeMixin, graphistry.PlotterBase.PlotterBase, object
```

### 1.3 graphistry.pygraphistry module

```
class graphistry.pygraphistry.NumpyJSONEncoder(*, skipkeys=False, ensure_ascii=True,
                                                check_circular=True, allow_nan=True,
                                                sort_keys=False, indent=None, separators=None, default=None)
```

Bases: json.encoder.JSONEncoder

**default** (obj)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

```
class graphistry.pygraphistry.PyGraphistry
```

Bases: object

**static addStyle** (bg=None, fg=None, logo=None, page=None)

Creates a base plotter with some style settings.

For parameters, see `plotter.addStyle`.

**Returns** Plotter

**Return type** *Plotter*

#### Example

```
import graphistry
graphistry.addStyle(bg={'color': 'black'})
```

**static** `api_key` (*value=None*)

Set or get the API key. Also set via environment variable GRAPHISTRY\_API\_KEY.

**static** `api_token` (*value=None*)

Set or get the API token. Also set via environment variable GRAPHISTRY\_API\_TOKEN.

**static** `api_token_refresh_ms` (*value=None*)

Set or get the API token refresh interval in milliseconds. None and 0 interpreted as no refreshing.

**static** `api_version` (*value=None*)

Set or get the API version: 1 or 2 for 1.0 (deprecated), 3 for 2.0 Also set via environment variable GRAPHISTRY\_API\_VERSION.

**static** `authenticate` ()

Authenticate via already provided configuration (api=1,2). This is called once automatically per session when uploading and rendering a visualization. In api=3, if token\_refresh\_ms > 0 (defaults to 10min), this starts an automatic refresh loop. In that case, note that a manual .login() is still required every 24hr by default.

**static** `bind` (*node=None, source=None, destination=None, edge\_title=None, edge\_label=None, edge\_color=None, edge\_weight=None, edge\_icon=None, edge\_size=None, edge\_opacity=None, edge\_source\_color=None, edge\_destination\_color=None, point\_title=None, point\_label=None, point\_color=None, point\_weight=None, point\_icon=None, point\_size=None, point\_opacity=None, point\_x=None, point\_y=None*)

Create a base plotter.

Typically called at start of a program. For parameters, see `plotter.bind()` .

**Returns** Plotter

**Return type** *Plotter*

#### Example

```
import graphistry
g = graphistry.bind()
```

**static** `bolt` (*driver=None*)

**Parameters** **driver** – Neo4j Driver or arguments for GraphDatabase.driver({...})

**Returns** Plotter w/neo4j

Call this to create a Plotter with an overridden neo4j driver.

#### Example

```
import graphistry
g = graphistry.bolt({ server: 'bolt://...', auth: ('<username>', '
↪<password>') })
```



```
import neo4j
import graphistry

driver = neo4j.GraphDatabase.driver(...)

g = graphistry.bolt(driver)
```

**static** `certificate_validation` (*value=None*)

Enable/Disable SSL certificate validation (True, False). Also set via environment variable GRAPHISTRY\_CERTIFICATE\_VALIDATION.

**static** `client_protocol_hostname` (*value=None*)

Get/set the client protocol+hostname for when display urls (distinct from uploading). Also set via environment variable GRAPHISTRY\_CLIENT\_PROTOCOL\_HOSTNAME. Defaults to hostname and no protocol (reusing environment protocol)

**static** `cosmos` (*COSMOS\_ACCOUNT=None*, *COSMOS\_DB=None*, *COSMOS\_CONTAINER=None*, *COSMOS\_PRIMARY\_KEY=None*, *gremlin\_client=None*)

Provide credentials as arguments, as environment variables, or by providing a gremlinpython client. Environment variable names are the same as the constructor argument names. If no client provided, create (connect)

#### Parameters

- **COSMOS\_ACCOUNT** (Optional[str]) – cosmos account
- **COSMOS\_DB** (Optional[str]) – cosmos db name
- **COSMOS\_CONTAINER** (Optional[str]) – cosmos container name
- **COSMOS\_PRIMARY\_KEY** (Optional[str]) – cosmos key
- **gremlin\_client** (Optional[Any]) – optional prebuilt client

**Return type** *Plotter*

**Returns** Plotter with data from a cypher query. This call binds *source*, *destination*, and *node*.

#### Example: Login and plot

```
import graphistry
(g = graphistry
 .cosmos(
     COSMOS_ACCOUNT='a',
     COSMOS_DB='b',
     COSMOS_CONTAINER='c',
     COSMOS_PRIMARY_KEY='d')
 .gremlin('g.E().sample(10)')
 .fetch_nodes() # Fetch properties for nodes
 .plot())
```

**static** `cypher` (*query*, *params={}*)

#### Parameters

- **query** – a cypher query
- **params** – cypher query arguments

**Returns** Plotter with data from a cypher query. This call binds *source*, *destination*, and *node*.

Call this to immediately execute a cypher query and store the graph in the resulting Plotter.

```
import graphistry
g = graphistry.bolt({ query='MATCH (a)-[r:PAYMENT]->(b) WHERE r.USD > 7000 AND r.USD < 10000 RETURN r ORDER BY r.USD DESC', params={
    "AccountId": 10 })
```

**static description** (*description*)

Upload description

**Parameters** *description* (*str*) – Upload description

**static drop\_graph** ()

Remove all graph nodes and edges from the database

**Return type** *Plotter*

**static edges** (*edges*, *source=None*, *destination=None*, *\*args*, *\*\*kwargs*)

Specify edge list data and associated edge attribute values. If a callable, will be called with current Plotter and whatever positional+named arguments

**Parameters** *edges* (*Pandas dataframe*, *NetworkX graph*, or *IGraph graph*) – Edges and their attributes, or transform from Plotter to edges

**Returns** Plotter

**Return type** *Plotter*

### Example

```
import graphistry
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
graphistry
    .bind(source='src', destination='dst')
    .edges(df)
    .plot()
```

### Example

```
import graphistry
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
graphistry
    .edges(df, 'src', 'dst')
    .plot()
```

### Example

```
:: import graphistry
```

```
def sample_edges(g, n): return g._edges.sample(n)
```

```
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
```

```
graphistry .edges(df, 'src', 'dst') .edges(sample_edges, n=2) .edges(sample_edges, None,
    None, 2) # equivalent .plot()
```

**static encode\_edge\_badge** (*column*, *position="TopRight"*, *categorical\_mapping=None*, *continuous\_binning=None*, *default\_mapping=None*, *comparator=None*, *color=None*, *bg=None*, *fg=None*, *for\_current=False*, *for\_default=True*, *as\_text=None*, *blend\_mode=None*, *style=None*, *border=None*, *shape=None*)

```
static encode_edge_color (column, palette=None, as_categorical=None,  
                           as_continuous=None, categorical_mapping=None, de-  
                           fault_mapping=None, for_default=True, for_current=False)
```

Set edge color with more control than bind()

#### Parameters

- **column** (*str*) – Data column name
- **palette** (*Optional[list]*) – Optional list of color-like strings. Ex: ["black", "#FF0", "rgb(255,255,255)"]. Used as a gradient for continuous and round-robin for categorical.
- **as\_categorical** (*Optional[bool]*) – Interpret column values as categorical. Ex: Uses palette via round-robin when more values than palette entries.
- **as\_continuous** (*Optional[bool]*) – Interpret column values as continuous. Ex: Uses palette for an interpolation gradient when more values than palette entries.
- **categorical\_mapping** (*Optional[dict]*) – Mapping from column values to color-like strings. Ex: {"car": "red", "truck": "#000"}
- **default\_mapping** (*Optional[str]*) – Augment categorical\_mapping with mapping for values not in categorical\_mapping. Ex: default\_mapping="gray".
- **for\_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for\_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

Returns Plotter

Return type *Plotter*

Example: See `encode_point_color`

```
static encode_edge_icon (column, categorical_mapping=None, continuous_binning=None,  
                           default_mapping=None, comparator=None, for_default=True,  
                           for_current=False, as_text=False, blend_mode=None, style=None,  
                           border=None, shape=None)
```

Set edge icon with more control than bind(). Values from Font Awesome 4 such as "laptop": <https://fontawesome.com/v4.7.0/icons/>

#### Parameters

- **column** (*str*) – Data column name
- **categorical\_mapping** (*Optional[dict]*) – Mapping from column values to icon name strings. Ex: {"toyota": "car", "ford": "truck"}
- **default\_mapping** (*Optional[Union[int, float]]*) – Augment categorical\_mapping with mapping for values not in categorical\_mapping. Ex: default\_mapping=50.
- **for\_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for\_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.
- **as\_text** (*Optional[bool]*) – Values should instead be treated as raw strings, instead of icons and images. (Default False.)
- **blend\_mode** (*Optional[str]*) – CSS blend mode

- **style** (*Optional[dict]*) – CSS filter properties - opacity, saturation, luminosity, grayscale, and more
- **border** (*Optional[dict]*) – Border properties - 'width', 'color', and 'stroke'

Returns Plotter

Return type *Plotter*

**Example: Set a string column of icons for the edge icons, same as bind(edge\_icon='my\_column')**

```
g2a = g.encode_edge_icon('my_icons_column')
```

**Example: Map specific values to specific icons, including with a default**

```
g2a = g.encode_edge_icon('brands', categorical_mapping={'toyota': 'car',
↳ 'ford': 'truck'})
g2b = g.encode_edge_icon('brands', categorical_mapping={'toyota': 'car',
↳ 'ford': 'truck'}, default_mapping='question')
```

**Example: Map countries to abbreviations**

```
g2a = g.encode_edge_icon('country_abbrev', as_text=True)
g2b = g.encode_edge_icon('country', categorical_mapping={'England': 'UK',
↳ 'America': 'US'}, default_mapping='')
```

**Example: Border**

```
g2b = g.encode_edge_icon('country', border={'width': 3, color: 'black',
↳ 'stroke': 'dashed'}, 'categorical_mapping={'England': 'UK', 'America':
↳ 'US'})
```

```
static encode_point_badge(column, position='TopRight', categorical_mapping=None,
continuous_binning=None, default_mapping=None, comparator=None, color=None, bg=None, fg=None, for_current=False,
for_default=True, as_text=None, blend_mode=None, style=None,
border=None, shape=None)
```

```
static encode_point_color(column, palette=None, as_categorical=None,
as_continuous=None, categorical_mapping=None, default_mapping=None, for_default=True, for_current=False)
```

Set point color with more control than bind()

#### Parameters

- **column** (*str*) – Data column name
- **palette** (*Optional[list]*) – Optional list of color-like strings. Ex: ["black", "#FF0", "rgb(255,255,255)"]. Used as a gradient for continuous and round-robin for categorical.
- **as\_categorical** (*Optional[bool]*) – Interpret column values as categorical. Ex: Uses palette via round-robin when more values than palette entries.
- **as\_continuous** (*Optional[bool]*) – Interpret column values as continuous. Ex: Uses palette for an interpolation gradient when more values than palette entries.
- **categorical\_mapping** (*Optional[dict]*) – Mapping from column values to color-like strings. Ex: {"car": "red", "truck": "#000"}

- **default\_mapping** (*Optional[str]*) – Augment categorical\_mapping with mapping for values not in categorical\_mapping. Ex: default\_mapping="gray".
- **for\_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for\_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

Returns Plotter

Return type *Plotter*

**Example: Set a palette-valued column for the color, same as bind(point\_color='my\_column')**

```
g2a = g.encode_point_color('my_int32_palette_column')
g2b = g.encode_point_color('my_int64_rgb_column')
```

**Example: Set a cold-to-hot gradient of along the spectrum blue, yellow, red**

```
g2 = g.encode_point_color('my_numeric_col', palette=["blue", "yellow",
↪ "red"], as_continuous=True)
```

**Example: Round-robin sample from 5 colors in hex format**

```
g2 = g.encode_point_color('my_distinctly_valued_col', palette=["#000", "
↪ #00F", "#0F0", "#0FF", "#FFF"], as_categorical=True)
```

**Example: Map specific values to specific colors, including with a default**

```
g2a = g.encode_point_color('brands', categorical_mapping={'toyota': 'red',
↪ 'ford': 'blue'})
g2a = g.encode_point_color('brands', categorical_mapping={'toyota': 'red',
↪ 'ford': 'blue'}, default_mapping='gray')
```

**static encode\_point\_icon**(*column*, *categorical\_mapping=None*, *continuous\_binning=None*, *default\_mapping=None*, *comparator=None*, *for\_default=True*, *for\_current=False*, *as\_text=False*, *blend\_mode=None*, *style=None*, *border=None*, *shape=None*)

Set node icon with more control than bind(). Values from Font Awesome 4 such as “laptop”: <https://fontawesome.com/v4.7.0/icons/>

#### Parameters

- **column** (*str*) – Data column name
- **categorical\_mapping** (*Optional[dict]*) – Mapping from column values to icon name strings. Ex: {"toyota": 'car', "ford": 'truck'}
- **default\_mapping** (*Optional[Union[int, float]]*) – Augment categorical\_mapping with mapping for values not in categorical\_mapping. Ex: default\_mapping=50.
- **for\_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for\_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.
- **as\_text** (*Optional[bool]*) – Values should instead be treated as raw strings, instead of icons and images. (Default False.)

- **blend\_mode** (*Optional[str]*) – CSS blend mode
- **style** (*Optional[dict]*) – CSS filter properties - opacity, saturation, luminosity, grayscale, and more
- **border** (*Optional[dict]*) – Border properties - ‘width’, ‘color’, and ‘stroke’

Returns Plotter

Return type *Plotter*

**Example:** Set a string column of icons for the point icons, same as `bind(point_icon='my_column')`

```
g2a = g.encode_point_icon('my_icons_column')
```

**Example:** Map specific values to specific icons, including with a default

```
g2a = g.encode_point_icon('brands', categorical_mapping={'toyota': 'car',  
↳ 'ford': 'truck'})  
g2b = g.encode_point_icon('brands', categorical_mapping={'toyota': 'car',  
↳ 'ford': 'truck'}, default_mapping='question')
```

**Example:** Map countries to abbreviations

```
g2b = g.encode_point_icon('country_abbrev', as_text=True)  
g2b = g.encode_point_icon('country', as_text=True, categorical_mapping={  
↳ 'England': 'UK', 'America': 'US'}, default_mapping='')
```

**Example:** Border

```
g2b = g.encode_point_icon('country', border={'width': 3, color: 'black',  
↳ 'stroke': 'dashed'}, categorical_mapping={'England': 'UK', 'America':  
↳ 'US'})
```

**static encode\_point\_size** (*column*, *categorical\_mapping=None*, *default\_mapping=None*,  
*for\_default=True, for\_current=False*)

Set point size with more control than `bind()`

**Parameters**

- **column** (*str*) – Data column name
- **categorical\_mapping** (*Optional[dict]*) – Mapping from column values to numbers. Ex: {“car”: 100, “truck”: 200}
- **default\_mapping** (*Optional[Union[int, float]]*) – Augment categorical\_mapping with mapping for values not in categorical\_mapping. Ex: default\_mapping=50.
- **for\_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for\_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

Returns Plotter

Return type *Plotter*

**Example:** Set a numerically-valued column for the size, same as `bind(point_size='my_column')`

```
g2a = g.encode_point_size('my_numeric_column')
```

**Example: Map specific values to specific colors, including with a default**

```
g2a = g.encode_point_size('brands', categorical_mapping={'toyota': 100,
↳ 'ford': 200})
g2b = g.encode_point_size('brands', categorical_mapping={'toyota': 100,
↳ 'ford': 200}, default_mapping=50)
```

**static graph** (*ig*)

**static gremlin** (*queries*)

Run one or more gremlin queries and get back the result as a graph object To support cosmosdb, sends as strings

**Example: Login and plot**

```
import graphistry
(graphistry
 .gremlin_client(my_gremlin_client)
 .gremlin('g.E().sample(10)')
 .fetch_nodes() # Fetch properties for nodes
 .plot())
```

**Parameters queries** (Union[str, Iterable[str]]) –

**Return type** Plottable

**static gremlin\_client** (*gremlin\_client=None*)

Pass in a generic gremlin python client

**Example: Login and plot**

```
import graphistry
from gremlin_python.driver.client import Client

my_gremlin_client = Client(
f'wss://MY_ACCOUNT.gremlin.cosmosdb.azure.com:443/',
'g',
username=f"/dbs/MY_DB/colls/{self.COSMOS_CONTAINER}",
password=self.COSMOS_PRIMARY_KEY,
message_serializer=GraphSONSerializersV2d0())

(graphistry
 .gremlin_client(my_gremlin_client)
 .gremlin('g.E().sample(10)')
 .fetch_nodes() # Fetch properties for nodes
 .plot())
```

**Parameters gremlin\_client** (Optional[Any]) –

**Return type** *Plotter*

**static gsql** (*query, bindings=None, dry\_run=False*)

Run Tigergraph query in interpreted mode and return transformed Plottable

**param query** Code to run

**type query** str

**param bindings** Mapping defining names of returned 'edges' and/or 'nodes', defaults to @@nodeList and @@edgeList

**type bindings** Optional[dict]

**param dry\_run** Return target URL without running

**type dry\_run** bool

**returns** Plotter

**rtype** Plotter

#### Example: Minimal

```
import graphistry
tg = graphistry.tigergraph()
tg.gsql("""
INTERPRET QUERY () FOR GRAPH Storage {

    OrAccum<BOOL> @@stop;
    ListAccum<EDGE> @@edgeList;
    SetAccum<vertex> @@set;

    @@set += to_vertex("61921", "Pool");

    Start = @@set;

    while Start.size() > 0 and @@stop == false do

        Start = select t from Start:s-(:e)-:t
        where e.goUpper == TRUE
        accum @@edgeList += e
        having t.type != "Service";
    end;

    print @@edgeList;
}
""").plot()
```

#### Example: Full

```
import graphistry
tg = graphistry.tigergraph()
tg.gsql("""
INTERPRET QUERY () FOR GRAPH Storage {

    OrAccum<BOOL> @@stop;
    ListAccum<EDGE> @@edgeList;
    SetAccum<vertex> @@set;

    @@set += to_vertex("61921", "Pool");

    Start = @@set;

    while Start.size() > 0 and @@stop == false do
```

(continues on next page)



(continued from previous page)

```

Start = select t from Start:s-(e)-:t
where e.goUpper == TRUE
accum @@edgeList += e
having t.type != "Service";
end;

print @@my_edge_list;
}
"""', {'edges': 'my_edge_list'}).plot()
```

**static gsql\_endpoint** (*self*, *method\_name*, *args*={}, *bindings*=None, *db*=None, *dry\_run*=False)

Invoke Tigergraph stored procedure at a user-defined endpoint and return transformed Plottable

#### Parameters

- **method\_name** (*str*) – Stored procedure name
- **args** (*Optional[dict]*) – Named endpoint arguments
- **bindings** (*Optional[dict]*) – Mapping defining names of returned ‘edges’ and/or ‘nodes’, defaults to @@nodeList and @@edgeList
- **db** (*Optional[str]*) – Name of the database, defaults to value set in .tigergraph(...)
- **dry\_run** (*bool*) – Return target URL without running

**Returns** Plotter

**Return type** *Plotter*

#### Example: Minimal

```

import graphistry
tg = graphistry.tigergraph(db='my_db')
tg.gsql_endpoint('neighbors').plot()
```

#### Example: Full

```

import graphistry
tg = graphistry.tigergraph()
tg.gsql_endpoint('neighbors', {'k': 2}, {'edges': 'my_edge_list'}, 'my_db'
→).plot()
```

#### Example: Read data

```

import graphistry
tg = graphistry.tigergraph()
out = tg.gsql_endpoint('neighbors')
(nodes_df, edges_df) = (out._nodes, out._edges)
```

**static hypergraph** (*raw\_events*, *entity\_types*=None, *opts*={}, *drop\_na*=True, *drop\_edge\_attrs*=False, *verbose*=True, *direct*=False, *engine*='pandas', *npartitions*=None, *chunksizes*=None)

Transform a dataframe into a hypergraph.

#### Parameters

- **raw\_events** (*pandas.DataFrame*) – Dataframe to transform (pandas or cudf).

- **entity\_types** (*Optional[list]*) – Columns (strings) to turn into nodes, None signifies all
- **opts** (*dict*) – See below
- **drop\_edge\_attrs** (*bool*) – Whether to include each row’s attributes on its edges, defaults to False (include)
- **verbose** (*bool*) – Whether to print size information
- **direct** (*bool*) – Omit hypernode and instead strongly connect nodes in an event
- **engine** (*bool*) – String (pandas, cudf, ...) for engine to use
- **npartitions** (*Optional[int]*) – For distributed engines, how many coarse-grained pieces to split events into
- **chunksize** (*Optional[int]*) – For distributed engines, split events after chunksize rows

Create a graph out of the dataframe, and return the graph components as dataframes, and the renderable result Plotter. Hypergraphs reveal relationships between rows and between column values. This transform is useful for lists of events, samples, relationships, and other structured high-dimensional data.

Specify local compute engine by passing *engine*='pandas', 'cudf', 'dask', 'dask\_cudf' (default: 'pandas'). If events are not in that engine’s format, they will be converted into it.

The transform creates a node for every unique value in the *entity\_types* columns (default: all columns). If *direct*=False (default), every row is also turned into a node. Edges are added to connect every table cell to its originating row’s node, or if *direct*=True, to the other nodes from the same row. Nodes are given the attribute 'type' corresponding to the originating column name, or in the case of a row, 'EventID'. Options further control the transform, such column category definitions for controlling whether values reoccurring in different columns should be treated as one node, or whether to only draw edges between certain column type pairs.

Consider a list of events. Each row represents a distinct event, and each column some metadata about an event. If multiple events have common metadata, they will be transitively connected through those metadata values. The layout algorithm will try to cluster the events together. Conversely, if an event has unique metadata, the unique metadata will turn into nodes that only have connections to the event node, and the clustering algorithm will cause them to form a ring around the event node.

Best practice is to set *EVENTID* to a row’s unique ID, *SKIP* to all non-categorical columns (or *entity\_types* to all categorical columns), and *CATEGORY* to group columns with the same kinds of values.

To prevent creating nodes for null values, set *drop\_na*=True. Some dataframe engines may have undesirable null handling, and recommend replacing None values with np.nan .

The optional *opts*={ ... } configuration options are:

- 'EVENTID': Column name to inspect for a row ID. By default, uses the row index.
- 'CATEGORIES': Dictionary mapping a category name to inhabiting columns. E.g., {'IP': ['srcAddress', 'dstAddress']}. If the same IP appears in both columns, this makes the transform generate one node for it, instead of one for each column.
- 'DELIM': When creating node IDs, defines the separator used between the column name and node value
- 'SKIP': List of column names to not turn into nodes. For example, dates and numbers are often skipped.
- 'EDGES': For *direct*=True, instead of making all edges, pick column pairs. E.g., {'a': ['b', 'd'], 'd': ['d']} creates edges between columns a->b and a->d, and self-edges d->d.

**Returns** {'entities': DF, 'events': DF, 'edges': DF, 'nodes': DF, 'graph': Plotter}

**Return type** dict

**Example: Connect user<-row->boss**

```
import graphistry
users_df = pd.DataFrame({'user': ['a', 'b', 'x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df)
g = h['graph'].plot()
```

**Example: Connect user->boss**

```
import graphistry
users_df = pd.DataFrame({'user': ['a', 'b', 'x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, direct=True)
g = h['graph'].plot()
```

**Example: Connect user<->boss**

```
import graphistry
users_df = pd.DataFrame({'user': ['a', 'b', 'x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, direct=True, opts={'EDGES': {'user': ['boss'], 'boss': ['user']}})
g = h['graph'].plot()
```

**Example: Only consider some columns for nodes**

```
import graphistry
users_df = pd.DataFrame({'user': ['a', 'b', 'x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, entity_types=['boss'])
g = h['graph'].plot()
```

**Example: Collapse matching user::<id> and boss::<id> nodes into one person::<id> node**

```
import graphistry
users_df = pd.DataFrame({'user': ['a', 'b', 'x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, opts={'CATEGORIES': {'person': ['user', 'boss']}})
g = h['graph'].plot()
```

**Example: Use cudf engine instead of pandas**

```
import cudf, graphistry
users_gdf = cudf.DataFrame({'user': ['a', 'b', 'x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_gdf, engine='cudf')
g = h['graph'].plot()
```

**Parameters**

- **entity\_types** (Optional[List[str]]) –
- **opts** (dict) –

- **drop\_na** (bool) –
- **drop\_edge\_attrs** (bool) –
- **verbose** (bool) –
- **direct** (bool) –
- **engine** (str) –
- **npartitions** (Optional[int]) –
- **chunksize** (Optional[int]) –

**static infer\_labels** (*self*)

**Returns** Plotter w/neo4j

- Prefers point\_title/point\_label if available
- Fallback to node id
- Raises exception if no nodes available, no likely candidates, and no matching node id fallback

#### Example

```
import graphistry
g = graphistry.nodes(pd.read_csv('nodes.csv'), 'id_col').infer_labels()
g.plot()
```

**static layout\_settings** (*play=None, locked\_x=None, locked\_y=None, locked\_r=None, left=None, top=None, right=None, bottom=None, lin\_log=None, strong\_gravity=None, dissuade\_hubs=None, edge\_influence=None, precision\_vs\_speed=None, gravity=None, scaling\_ratio=None*)

Set layout options. Additive over previous settings.

Corresponds to options at <https://hub.graphistry.com/docs/api/1/rest/url/#urloptions>

#### Example: Animated radial layout

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a', 'b', 'c', 'd'], 'boss': ['c', 'c', 'e', 'e', 'e']})
nodes = pd.DataFrame({
    'n': ['a', 'b', 'c', 'd', 'e'],
    'y': [1, 1, 2, 3, 4],
    'x': [1, 1, 0, 0, 0],
})
g = (graphistry
     .edges(edges, 's', 'd')
     .nodes(nodes, 'n')
     .layout_settings(locked_r=True, play=2000)
g.plot())
```

#### Parameters

- **play** (Optional[int]) –
- **locked\_x** (Optional[bool]) –
- **locked\_y** (Optional[bool]) –
- **locked\_r** (Optional[bool]) –

- **left** (Optional[float]) –
- **top** (Optional[float]) –
- **right** (Optional[float]) –
- **bottom** (Optional[float]) –
- **lin\_log** (Optional[bool]) –
- **strong\_gravity** (Optional[bool]) –
- **dissuade\_hubs** (Optional[bool]) –
- **edge\_influence** (Optional[float]) –
- **precision\_vs\_speed** (Optional[float]) –
- **gravity** (Optional[float]) –
- **scaling\_ratio** (Optional[float]) –

**static login** (*username, password, fail\_silent=False*)

Authenticate and set token for reuse (api=3). If token\_refresh\_ms (default: 10min), auto-refreshes token. By default, must be reinvoked within 24hr.

**static name** (*name*)

Upload name

**Parameters** **name** (*str*) – Upload name

**static neptune** (*NEPTUNE\_READER\_HOST=None, NEPTUNE\_READER\_PORT=None, NEPTUNE\_READER\_PROTOCOL='wss', endpoint=None, gremlin\_client=None*)

Provide credentials as arguments, as environment variables, or by providing a gremlinpython client Environment variable names are the same as the constructor argument names If endpoint provided, do not need host/port/protocol If no client provided, create (connect)

#### Example: Login and plot via parrams

```
import graphistry
(graphistry
 .neptune(
     NEPTUNE_READER_PROTOCOL='wss'
     NEPTUNE_READER_HOST='neptunedbcluster-xyz.cluster-ro-abc.us-
↪east-1.neptune.amazonaws.com'
     NEPTUNE_READER_PORT='8182'
 )
 .gremlin('g.E().sample(10)')
 .fetch_nodes() # Fetch properties for nodes
 .plot())
```

#### Example: Login and plot via env vars

```
import graphistry
(graphistry
 .neptune()
 .gremlin('g.E().sample(10)')
 .fetch_nodes() # Fetch properties for nodes
 .plot())
```

#### Example: Login and plot via endpoint

```
import graphistry
(graphistry
 .neptune(endpoint='wss://neptunedbcluster-xyz.cluster-ro-abc.us-
 →east-1.neptune.amazonaws.com:8182/gremlin')
 .gremlin('g.E().sample(10)')
 .fetch_nodes() # Fetch properties for nodes
 .plot())
```

#### Example: Login and plot via client

```
import graphistry
(graphistry
 .neptune(gremlin_client=client)
 .gremlin('g.E().sample(10)')
 .fetch_nodes() # Fetch properties for nodes
 .plot())
```

#### Parameters

- **NEPTUNE\_READER\_HOST** (Optional[str]) –
- **NEPTUNE\_READER\_PORT** (Optional[str]) –
- **NEPTUNE\_READER\_PROTOCOL** (Optional[str]) –
- **endpoint** (Optional[str]) –
- **gremlin\_client** (Optional[Any]) –

Return type *Plotter*

**static nodes** (*nodes*, *node=None*, *\*args*, *\*\*kwargs*)

Specify the set of nodes and associated data. If a callable, will be called with current Plotter and whatever positional+named arguments

Must include any nodes referenced in the edge list.

**Parameters nodes** (*Pandas dataframe or Callable*) – Nodes and their attributes.

**Returns** Plotter

Return type *Plotter*

#### Example

```
import graphistry

es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g = graphistry
 .bind(source='src', destination='dst')
 .edges(es)

vs = pandas.DataFrame({'v': [0,1,2], 'lbl': ['a', 'b', 'c']})
g = g.bind(node='v').nodes(vs)

g.plot()
```

#### Example

```
import graphistry

es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g = graphistry.edges(es, 'src', 'dst')

vs = pandas.DataFrame({'v': [0,1,2], 'lbl': ['a', 'b', 'c']})
g = g.nodes(vs, 'v')

g.plot()
```

**Example**

```
:: import graphistry

def sample_nodes(g, n): return g._nodes.sample(n)

df = pandas.DataFrame({'id': [0,1,2], 'v': [1,2,0]})

graphistry .nodes(df, 'id') ..nodes(sample_nodes, n=2) ..nodes(sample_nodes, None, 2) # equivalent
.plot()
```

**static** `nodexl(xls_or_url, source='default', engine=None, verbose=False)`

**Parameters**

- **xls\_or\_url** – file/http path string to a nodexl-generated xls, or a pandas ExcelFile() object
- **source** – optionally activate binding by string name for a known nodexl data source ('twitter', 'wikimedia')
- **engine** – optionally set a pandas Excel engine
- **verbose** – optionally enable printing progress by overriding to True

**static** `not_implemented_thunk()`

**static** `pipe(graph_transform, *args, **kwargs)`

Create new Plotter derived from current

**Parameters** `graph_transform(Callable)` –

**Example: Simple**

```
import graphistry

def fill_missing_bindings(g, source='src', destination='dst'):
    return g.bind(source=source, destination=destination)

graphistry
    .edges(pandas.DataFrame({'src': [0,1,2], 'd': [1,2,0]}))
    .pipe(fill_missing_bindings, destination='d') # binds 'src'
    .plot()
```

**Return type** `Plottable`

**static** `protocol(value=None)`

Set or get the protocol ('http' or 'https'). Set automatically when using a server alias. Also set via environment variable GRAPHISTRY\_PROTOCOL.

**static refresh** (*token=None, fail\_silent=False*)

Use self or provided JWT token to get a fresher one. If self token, internalize upon refresh.

**static register** (*key=None, username=None, password=None, token=None, server=None, protocol=None, api=None, certificate\_validation=None, bolt=None, token\_refresh\_ms=600000, store\_token\_creds\_in\_memory=None, client\_protocol\_hostname=None*)

API key registration and server selection

Changing the key effects all derived Plotter instances.

Provide one of key (api=1,2) or username/password (api=3) or token (api=3).

#### Parameters

- **key** (*Optional[str]*) – API key (1.0 API).
- **username** (*Optional[str]*) – Account username (2.0 API).
- **password** (*Optional[str]*) – Account password (2.0 API).
- **token** (*Optional[str]*) – Valid Account JWT token (2.0). Provide token, or username/password, but not both.
- **server** (*Optional[str]*) – URL of the visualization server.
- **certificate\_validation** (*Optional[bool]*) – Override default-on check for valid TLS certificate by setting to True.
- **bolt** (*Union[dict, Any]*) – Neo4j bolt information. Optional driver or named constructor arguments for instantiating a new one.
- **protocol** (*Optional[str]*) – Protocol used to contact visualization server, defaults to “https”.
- **token\_refresh\_ms** (*int*) – Ignored for now; JWT token auto-refreshed on plot() calls.
- **store\_token\_creds\_in\_memory** (*Optional[bool]*) – Store username/password in-memory for JWT token refreshes (Token-originated have a hard limit, so always-on requires creds somewhere)
- **client\_protocol\_hostname** (*Optional[str]*) – Override protocol and host shown in browser. Defaults to protocol/server or envvar GRAPHISTRY\_CLIENT\_PROTOCOL\_HOSTNAME.

Returns None.

Return type None

#### Example: Standard (2.0 api by username/password)

```
import graphistry
graphistry.register(api=3, protocol='http', server='200.1.1.1', username=
↳ 'person', password='pwd')
```

#### Example: Standard (2.0 api by token)

```
import graphistry
graphistry.register(api=3, protocol='http', server='200.1.1.1', token='abc
↳ ')
```

#### Example: Remote browser to Graphistry-provided notebook server (2.0)



```
import graphistry
graphistry.register(api=3, protocol='http', server='nginx', client_
↳protocol_hostname='https://my.site.com', token='abc')
```

**Example: Standard (1.0)**

```
import graphistry
graphistry.register(api=1, key="my api key")
```

**relogin()****static server** (*value=None*)

Get the hostname of the server or set the server using hostname or aliases. Also set via environment variable GRAPHISTRY\_HOSTNAME.

**static set\_bolt\_driver** (*driver=None*)**static settings** (*height=None, url\_params={}, render=None*)**static store\_token\_creds\_in\_memory** (*value=None*)

Cache credentials for JWT token access. Default off due to not being safe.

**static style** (*bg=None, fg=None, logo=None, page=None*)

Creates a base plotter with some style settings.

For parameters, see `plotter.style`.

**Returns** Plotter**Return type** *Plotter***Example**

```
import graphistry
graphistry.style(bg={'color': 'black'})
```

**static tigergraph** (*protocol='http', server='localhost', web\_port=14240, api\_port=9000, db=None, user='tigergraph', pwd='tigergraph', verbose=False*)

Register Tigergraph connection setting defaults

**Parameters**

- **protocol** (*Optional[str]*) – Protocol used to contact the database.
- **server** (*Optional[str]*) – Domain of the database
- **web\_port** (*Optional[int]*) –
- **api\_port** (*Optional[int]*) –
- **db** (*Optional[str]*) – Name of the database
- **user** (*Optional[str]*) –
- **pwd** (*Optional[str]*) –
- **verbose** (*Optional[bool]*) – Whether to print operations

**Returns** Plotter**Return type** *Plotter***Example: Standard**

```
import graphistry
tg = graphistry.tigergraph(protocol='https', server='acme.com', db='my_db
↪', user='alice', pwd='tigergraph2')
```

**static verify\_token** (*token=None, fail\_silent=False*)

Return True iff current or provided token is still valid

**Return type** bool

graphistry.pygraphistry.**addStyle** (*bg=None, fg=None, logo=None, page=None*)

Creates a base plotter with some style settings.

For parameters, see `plotter.addStyle`.

**Returns** Plotter

**Return type** *Plotter*

**Example**

```
import graphistry
graphistry.addStyle(bg={'color': 'black'})
```

graphistry.pygraphistry.**api\_token** (*value=None*)

Set or get the API token. Also set via environment variable GRAPHISTRY\_API\_TOKEN.

graphistry.pygraphistry.**bind** (*node=None, source=None, destination=None, edge\_title=None,*  
*edge\_label=None, edge\_color=None, edge\_weight=None,*  
*edge\_icon=None, edge\_size=None, edge\_opacity=None,*  
*edge\_source\_color=None, edge\_destination\_color=None,*  
*point\_title=None, point\_label=None, point\_color=None,*  
*point\_weight=None, point\_icon=None, point\_size=None,*  
*point\_opacity=None, point\_x=None, point\_y=None*)

Create a base plotter.

Typically called at start of a program. For parameters, see `plotter.bind()` .

**Returns** Plotter

**Return type** *Plotter*

**Example**

```
import graphistry
g = graphistry.bind()
```

graphistry.pygraphistry.**bolt** (*driver=None*)

**Parameters** **driver** – Neo4j Driver or arguments for `GraphDatabase.driver({...})`

**Returns** Plotter w/neo4j

Call this to create a Plotter with an overridden neo4j driver.

**Example**

```
import graphistry
g = graphistry.bolt({ server: 'bolt://...', auth: ('<username>', '
↪<password>') })
```

```
import neo4j
import graphistry

driver = neo4j.GraphDatabase.driver(...)

g = graphistry.bolt(driver)
```

graphistry.pygraphistry.**client\_protocol\_hostname** (*value=None*)

Get/set the client protocol+hostname for when display urls (distinct from uploading). Also set via environment variable GRAPHISTRY\_CLIENT\_PROTOCOL\_HOSTNAME. Defaults to hostname and no protocol (reusing environment protocol)

graphistry.pygraphistry.**cosmos** (*COSMOS\_ACCOUNT=None, COSMOS\_DB=None, COSMOS\_CONTAINER=None, COSMOS\_PRIMARY\_KEY=None, gremlin\_client=None*)

Provide credentials as arguments, as environment variables, or by providing a gremlinpython client Environment variable names are the same as the constructor argument names If no client provided, create (connect)

#### Parameters

- **COSMOS\_ACCOUNT** (Optional[str]) – cosmos account
- **COSMOS\_DB** (Optional[str]) – cosmos db name
- **COSMOS\_CONTAINER** (Optional[str]) – cosmos container name
- **COSMOS\_PRIMARY\_KEY** (Optional[str]) – cosmos key
- **gremlin\_client** (Optional[Any]) – optional prebuilt client

**Return type** *Plotter*

**Returns** Plotter with data from a cypher query. This call binds *source*, *destination*, and *node*.

#### Example: Login and plot

```
import graphistry
(g = graphistry
 .cosmos(
     COSMOS_ACCOUNT='a',
     COSMOS_DB='b',
     COSMOS_CONTAINER='c',
     COSMOS_PRIMARY_KEY='d')
 .gremlin('g.E().sample(10)')
 .fetch_nodes() # Fetch properties for nodes
 .plot())
```

graphistry.pygraphistry.**cypher** (*query, params={}*)

#### Parameters

- **query** – a cypher query
- **params** – cypher query arguments

**Returns** Plotter with data from a cypher query. This call binds *source*, *destination*, and *node*.

Call this to immediately execute a cypher query and store the graph in the resulting Plotter.

```
import graphistry
g = graphistry.bolt({ query='MATCH (a)-[r:PAYMENT]->(b) WHERE r.USD > 7000 AND r.USD < 10000 RETURN r ORDER BY r.USD DESC', params={
    "AccountId": 10 }})
```

`graphistry.pygraphistry.description(description)`

Upload description

**Parameters** `description` (*str*) – Upload description

`graphistry.pygraphistry.drop_graph()`

Remove all graph nodes and edges from the database

**Return type** *Plotter*

`graphistry.pygraphistry.edges(edges, source=None, destination=None, *args, **kwargs)`

Specify edge list data and associated edge attribute values. If a callable, will be called with current Plotter and whatever positional+named arguments

**Parameters** `edges` (*Pandas dataframe, NetworkX graph, or IGraph graph*) – Edges and their attributes, or transform from Plotter to edges

**Returns** Plotter

**Return type** *Plotter*

## Example

```
import graphistry
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
graphistry
    .bind(source='src', destination='dst')
    .edges(df)
    .plot()
```

## Example

```
import graphistry
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
graphistry
    .edges(df, 'src', 'dst')
    .plot()
```

## Example

```
:: import graphistry

def sample_edges(g, n): return g._edges.sample(n)

df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})

graphistry .edges(df, 'src', 'dst') .edges(sample_edges, n=2) .edges(sample_edges, None, None, 2)
# equivalent .plot()
```

`graphistry.pygraphistry.encode_edge_badge` (*column, position='TopRight', categorical\_mapping=None, continuous\_binning=None, default\_mapping=None, comparator=None, color=None, bg=None, fg=None, for\_current=False, for\_default=True, as\_text=None, blend\_mode=None, style=None, border=None, shape=None*)

`graphistry.pygraphistry.encode_edge_color` (*column, palette=None, as\_categorical=None, as\_continuous=None, categorical\_mapping=None, default\_mapping=None, for\_default=True, for\_current=False*)

Set edge color with more control than bind()

**Parameters**

- **column** (*str*) – Data column name
- **palette** (*Optional[list]*) – Optional list of color-like strings. Ex: ["black", "#FF0", "rgb(255,255,255)"]. Used as a gradient for continuous and round-robin for categorical.
- **as\_categorical** (*Optional[bool]*) – Interpret column values as categorical. Ex: Uses palette via round-robin when more values than palette entries.
- **as\_continuous** (*Optional[bool]*) – Interpret column values as continuous. Ex: Uses palette for an interpolation gradient when more values than palette entries.
- **categorical\_mapping** (*Optional[dict]*) – Mapping from column values to color-like strings. Ex: {"car": "red", "truck": "#000"}
- **default\_mapping** (*Optional[str]*) – Augment categorical\_mapping with mapping for values not in categorical\_mapping. Ex: default\_mapping="gray".
- **for\_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for\_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

**Returns** Plotter**Return type** *Plotter***Example:** See `encode_point_color`

```
graphistry.pygraphistry.encode_edge_icon(column, categorical_mapping=None, continuous_binning=None, default_mapping=None, comparator=None, for_default=True, for_current=False, as_text=False, blend_mode=None, style=None, border=None, shape=None)
```

Set edge icon with more control than `bind()`. Values from Font Awesome 4 such as "laptop": <https://fontawesome.com/v4.7.0/icons/>

**Parameters**

- **column** (*str*) – Data column name
- **categorical\_mapping** (*Optional[dict]*) – Mapping from column values to icon name strings. Ex: {"toyota": 'car', "ford": 'truck'}
- **default\_mapping** (*Optional[Union[int, float]]*) – Augment categorical\_mapping with mapping for values not in categorical\_mapping. Ex: default\_mapping=50.
- **for\_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for\_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.
- **as\_text** (*Optional[bool]*) – Values should instead be treated as raw strings, instead of icons and images. (Default False.)
- **blend\_mode** (*Optional[str]*) – CSS blend mode
- **style** (*Optional[dict]*) – CSS filter properties - opacity, saturation, luminosity, grayscale, and more

- **border** (*Optional[dict]*) – Border properties - ‘width’, ‘color’, and ‘stroke’

**Returns** Plotter

**Return type** *Plotter*

**Example:** Set a string column of icons for the edge icons, same as `bind(edge_icon='my_column')`

```
g2a = g.encode_edge_icon('my_icons_column')
```

**Example:** Map specific values to specific icons, including with a default

```
g2a = g.encode_edge_icon('brands', categorical_mapping={'toyota': 'car', 'ford': 'truck'})
g2b = g.encode_edge_icon('brands', categorical_mapping={'toyota': 'car', 'ford': 'truck'}, default_mapping='question')
```

**Example:** Map countries to abbreviations

```
g2a = g.encode_edge_icon('country_abbrev', as_text=True)
g2b = g.encode_edge_icon('country', categorical_mapping={'England': 'UK', 'America': 'US'}, default_mapping='')
```

**Example:** Border

```
g2b = g.encode_edge_icon('country', border={'width': 3, color: 'black', 'stroke': 'dashed'}, categorical_mapping={'England': 'UK', 'America': 'US'})
```

```
graphistry.pygraphistry.encode_point_badge(column, position='TopRight', categorical_mapping=None, continuous_binning=None, default_mapping=None, comparator=None, color=None, bg=None, fg=None, for_current=False, for_default=True, as_text=None, blend_mode=None, style=None, border=None, shape=None)

graphistry.pygraphistry.encode_point_color(column, palette=None, as_categorical=None, as_continuous=None, categorical_mapping=None, default_mapping=None, for_default=True, for_current=False)
```

Set point color with more control than `bind()`

#### Parameters

- **column** (*str*) – Data column name
- **palette** (*Optional[list]*) – Optional list of color-like strings. Ex: [“black”, “#FF0”, “rgb(255,255,255)”]. Used as a gradient for continuous and round-robin for categorical.
- **as\_categorical** (*Optional[bool]*) – Interpret column values as categorical. Ex: Uses palette via round-robin when more values than palette entries.
- **as\_continuous** (*Optional[bool]*) – Interpret column values as continuous. Ex: Uses palette for an interpolation gradient when more values than palette entries.
- **categorical\_mapping** (*Optional[dict]*) – Mapping from column values to color-like strings. Ex: {“car”: “red”, “truck”: #000}

- **default\_mapping** (*Optional[str]*) – Augment categorical\_mapping with mapping for values not in categorical\_mapping. Ex: default\_mapping="gray".
- **for\_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for\_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

**Returns** Plotter

**Return type** *Plotter*

**Example:** Set a palette-valued column for the color, same as bind(point\_color='my\_column')

```
g2a = g.encode_point_color('my_int32_palette_column')
g2b = g.encode_point_color('my_int64_rgb_column')
```

**Example:** Set a cold-to-hot gradient of along the spectrum blue, yellow, red

```
g2 = g.encode_point_color('my_numeric_col', palette=["blue", "yellow", "red"],
↪ as_continuous=True)
```

**Example:** Round-robin sample from 5 colors in hex format

```
g2 = g.encode_point_color('my_distinctly_valued_col', palette=["#000", "#00F",
↪ "#0F0", "#0FF", "#FFF"], as_categorical=True)
```

**Example:** Map specific values to specific colors, including with a default

```
g2a = g.encode_point_color('brands', categorical_mapping={'toyota': 'red',
↪ 'ford': 'blue'})
g2a = g.encode_point_color('brands', categorical_mapping={'toyota': 'red',
↪ 'ford': 'blue'}, default_mapping='gray')
```

graphistry.pygraphistry.**encode\_point\_icon**(column, categorical\_mapping=None, continuous\_binning=None, default\_mapping=None, comparator=None, for\_default=True, for\_current=False, as\_text=False, blend\_mode=None, style=None, border=None, shape=None)

Set node icon with more control than bind(). Values from Font Awesome 4 such as “laptop”: <https://fontawesome.com/v4.7.0/icons/>

#### Parameters

- **column** (*str*) – Data column name
- **categorical\_mapping** (*Optional[dict]*) – Mapping from column values to icon name strings. Ex: {"toyota": "car", "ford": "truck"}
- **default\_mapping** (*Optional[Union[int, float]]*) – Augment categorical\_mapping with mapping for values not in categorical\_mapping. Ex: default\_mapping=50.
- **for\_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for\_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

- **as\_text** (*Optional[bool]*) – Values should instead be treated as raw strings, instead of icons and images. (Default False.)
- **blend\_mode** (*Optional[str]*) – CSS blend mode
- **style** (*Optional[dict]*) – CSS filter properties - opacity, saturation, luminosity, grayscale, and more
- **border** (*Optional[dict]*) – Border properties - 'width', 'color', and 'stroke'

**Returns** Plotter

**Return type** *Plotter*

**Example: Set a string column of icons for the point icons, same as `bind(point_icon='my_column')`**

```
g2a = g.encode_point_icon('my_icons_column')
```

**Example: Map specific values to specific icons, including with a default**

```
g2a = g.encode_point_icon('brands', categorical_mapping={'toyota': 'car',
↳ 'ford': 'truck'})
g2b = g.encode_point_icon('brands', categorical_mapping={'toyota': 'car',
↳ 'ford': 'truck'}, default_mapping='question')
```

**Example: Map countries to abbreviations**

```
g2b = g.encode_point_icon('country_abbrev', as_text=True)
g2b = g.encode_point_icon('country', as_text=True, categorical_mapping={
↳ 'England': 'UK', 'America': 'US'}, default_mapping='')
```

**Example: Border**

```
g2b = g.encode_point_icon('country', border={'width': 3, color: 'black',
↳ 'stroke': 'dashed'}, 'categorical_mapping={'England': 'UK', 'America': 'US'})
↳ )
```

`graphistry.pygraphistry.encode_point_size` (*column*, *categorical\_mapping=None*, *default\_mapping=None*, *for\_default=True*, *for\_current=False*)

Set point size with more control than `bind()`

**Parameters**

- **column** (*str*) – Data column name
- **categorical\_mapping** (*Optional[dict]*) – Mapping from column values to numbers. Ex: {"car": 100, "truck": 200}
- **default\_mapping** (*Optional[Union[int, float]]*) – Augment categorical\_mapping with mapping for values not in categorical\_mapping. Ex: default\_mapping=50.
- **for\_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for\_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

**Returns** Plotter

**Return type** *Plotter*



**Example: Set a numerically-valued column for the size, same as bind(point\_size='my\_column')**

```
g2a = g.encode_point_size('my_numeric_column')
```

**Example: Map specific values to specific colors, including with a default**

```
g2a = g.encode_point_size('brands', categorical_mapping={'toyota': 100, 'ford': 200})
g2b = g.encode_point_size('brands', categorical_mapping={'toyota': 100, 'ford': 200}, default_mapping=50)
```

graphistry.pygraphistry.**graph** (ig)

graphistry.pygraphistry.**gremlin** (queries)

Run one or more gremlin queries and get back the result as a graph object To support cosmosdb, sends as strings

**Example: Login and plot**

```
import graphistry
(graphistry
 .gremlin_client(my_gremlin_client)
 .gremlin('g.E().sample(10)')
 .fetch_nodes() # Fetch properties for nodes
 .plot())
```

**Parameters** **queries** (Union[str, Iterable[str]]) –

**Return type** Plottable

graphistry.pygraphistry.**gremlin\_client** (gremlin\_client=None)

Pass in a generic gremlin python client

**Example: Login and plot**

```
import graphistry
from gremlin_python.driver.client import Client

my_gremlin_client = Client(
f'wss://MY_ACCOUNT.gremlin.cosmosdb.azure.com:443/',
'g',
username=f"/dbs/MY_DB/colls/{self.COSMOS_CONTAINER}",
password=self.COSMOS_PRIMARY_KEY,
message_serializer=GraphSONSerializersV2d0())

(graphistry
 .gremlin_client(my_gremlin_client)
 .gremlin('g.E().sample(10)')
 .fetch_nodes() # Fetch properties for nodes
 .plot())
```

**Parameters** **gremlin\_client** (Optional[Any]) –

**Return type** *Plotter*

graphistry.pygraphistry.**gsq1** (query, bindings=None, dry\_run=False)

Run Tigergraph query in interpreted mode and return transformed Plottable

**param** query Code to run

**type query** str

**param bindings** Mapping defining names of returned 'edges' and/or 'nodes', defaults to @@nodeList and @@edgeList

**type bindings** Optional[dict]

**param dry\_run** Return target URL without running

**type dry\_run** bool

**returns** Plotter

**rtype** Plotter

### Example: Minimal

```
import graphistry
tg = graphistry.tigergraph()
tg.gsql("""
INTERPRET QUERY () FOR GRAPH Storage {

    OrAccum<BOOL> @@stop;
    ListAccum<EDGE> @@edgeList;
    SetAccum<vertex> @@set;

    @@set += to_vertex("61921", "Pool");

    Start = @@set;

    while Start.size() > 0 and @@stop == false do

        Start = select t from Start:s-(:e)-:t
        where e.goUpper == TRUE
        accum @@edgeList += e
        having t.type != "Service";
        end;

        print @@edgeList;
    }
""").plot()
```

### Example: Full

```
import graphistry
tg = graphistry.tigergraph()
tg.gsql("""
INTERPRET QUERY () FOR GRAPH Storage {

    OrAccum<BOOL> @@stop;
    ListAccum<EDGE> @@edgeList;
    SetAccum<vertex> @@set;

    @@set += to_vertex("61921", "Pool");

    Start = @@set;

    while Start.size() > 0 and @@stop == false do
```

(continues on next page)

(continued from previous page)

```

Start = select t from Start:s-(:e)-:t
where e.goUpper == TRUE
accum @@edgeList += e
having t.type != "Service";
end;

print @@my_edge_list;
}
"""', {'edges': 'my_edge_list'}).plot()

```

`graphistry.pygraphistry.gsql_endpoint` (*self*, *method\_name*, *args*={}, *bindings*=None, *db*=None, *dry\_run*=False)

Invoke Tigergraph stored procedure at a user-defined endpoint and return transformed Plottable

#### Parameters

- **method\_name** (*str*) – Stored procedure name
- **args** (*Optional[dict]*) – Named endpoint arguments
- **bindings** (*Optional[dict]*) – Mapping defining names of returned ‘edges’ and/or ‘nodes’, defaults to @@nodeList and @@edgeList
- **db** (*Optional[str]*) – Name of the database, defaults to value set in `.tigergraph(...)`
- **dry\_run** (*bool*) – Return target URL without running

**Returns** Plotter

**Return type** *Plotter*

#### Example: Minimal

```

import graphistry
tg = graphistry.tigergraph(db='my_db')
tg.gsql_endpoint('neighbors').plot()

```

#### Example: Full

```

import graphistry
tg = graphistry.tigergraph()
tg.gsql_endpoint('neighbors', {'k': 2}, {'edges': 'my_edge_list'}, 'my_db').
    plot()

```

#### Example: Read data

```

import graphistry
tg = graphistry.tigergraph()
out = tg.gsql_endpoint('neighbors')
(nodes_df, edges_df) = (out._nodes, out._edges)

```

`graphistry.pygraphistry.hypergraph` (*raw\_events*, *entity\_types*=None, *opts*={}, *drop\_na*=True, *drop\_edge\_attrs*=False, *verbose*=True, *direct*=False, *engine*='pandas', *npartitions*=None, *chunks*=None)

Transform a dataframe into a hypergraph.

#### Parameters

- **raw\_events** (*pandas.DataFrame*) – Dataframe to transform (pandas or cudf).

- **entity\_types** (*Optional[list]*) – Columns (strings) to turn into nodes, None signifies all
- **opts** (*dict*) – See below
- **drop\_edge\_attrs** (*bool*) – Whether to include each row’s attributes on its edges, defaults to False (include)
- **verbose** (*bool*) – Whether to print size information
- **direct** (*bool*) – Omit hypernode and instead strongly connect nodes in an event
- **engine** (*bool*) – String (pandas, cudf, ...) for engine to use
- **npartitions** (*Optional[int]*) – For distributed engines, how many coarse-grained pieces to split events into
- **chunksize** (*Optional[int]*) – For distributed engines, split events after chunksize rows

Create a graph out of the dataframe, and return the graph components as dataframes, and the renderable result Plotter. Hypergraphs reveal relationships between rows and between column values. This transform is useful for lists of events, samples, relationships, and other structured high-dimensional data.

Specify local compute engine by passing *engine='pandas', 'cudf', 'dask', 'dask\_cudf'* (default: 'pandas'). If events are not in that engine’s format, they will be converted into it.

The transform creates a node for every unique value in the *entity\_types* columns (default: all columns). If *direct=False* (default), every row is also turned into a node. Edges are added to connect every table cell to its originating row’s node, or if *direct=True*, to the other nodes from the same row. Nodes are given the attribute 'type' corresponding to the originating column name, or in the case of a row, 'EventID'. Options further control the transform, such column category definitions for controlling whether values reoccurring in different columns should be treated as one node, or whether to only draw edges between certain column type pairs.

Consider a list of events. Each row represents a distinct event, and each column some metadata about an event. If multiple events have common metadata, they will be transitively connected through those metadata values. The layout algorithm will try to cluster the events together. Conversely, if an event has unique metadata, the unique metadata will turn into nodes that only have connections to the event node, and the clustering algorithm will cause them to form a ring around the event node.

Best practice is to set *EVENTID* to a row’s unique ID, *SKIP* to all non-categorical columns (or *entity\_types* to all categorical columns), and *CATEGORY* to group columns with the same kinds of values.

To prevent creating nodes for null values, set *drop\_na=True*. Some dataframe engines may have undesirable null handling, and recommend replacing None values with *np.nan*.

The optional *opts={...}* configuration options are:

- 'EVENTID': Column name to inspect for a row ID. By default, uses the row index.
- 'CATEGORIES': Dictionary mapping a category name to inhabiting columns. E.g., {'IP': ['srcAddress', 'dstAddress']}. If the same IP appears in both columns, this makes the transform generate one node for it, instead of one for each column.
- 'DELIM': When creating node IDs, defines the separator used between the column name and node value
- 'SKIP': List of column names to not turn into nodes. For example, dates and numbers are often skipped.
- 'EDGES': For *direct=True*, instead of making all edges, pick column pairs. E.g., {'a': ['b', 'd'], 'd': ['d']} creates edges between columns *a->b* and *a->d*, and self-edges *d->d*.

**Returns** {'entities': DF, 'events': DF, 'edges': DF, 'nodes': DF, 'graph': Plotter}

**Return type** dict

**Example: Connect user<-row->boss**

```
import graphistry
users_df = pd.DataFrame({'user': ['a', 'b', 'x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df)
g = h['graph'].plot()
```

**Example: Connect user->boss**

```
import graphistry
users_df = pd.DataFrame({'user': ['a', 'b', 'x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, direct=True)
g = h['graph'].plot()
```

**Example: Connect user<->boss**

```
import graphistry
users_df = pd.DataFrame({'user': ['a', 'b', 'x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, direct=True, opts={'EDGES': {'user': [
↔ 'boss'], 'boss': ['user']}})
g = h['graph'].plot()
```

**Example: Only consider some columns for nodes**

```
import graphistry
users_df = pd.DataFrame({'user': ['a', 'b', 'x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, entity_types=['boss'])
g = h['graph'].plot()
```

**Example: Collapse matching user::<id> and boss::<id> nodes into one person::<id> node**

```
import graphistry
users_df = pd.DataFrame({'user': ['a', 'b', 'x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, opts={'CATEGORIES': {'person': ['user
↔ ', 'boss']}})
g = h['graph'].plot()
```

**Example: Use cudf engine instead of pandas**

```
import cudf, graphistry
users_gdf = cudf.DataFrame({'user': ['a', 'b', 'x'], 'boss': ['x', 'x', 'y
↔']})
h = graphistry.hypergraph(users_gdf, engine='cudf')
g = h['graph'].plot()
```

**Parameters**

- **entity\_types** (Optional[List[str]]) –
- **opts** (dict) –
- **drop\_na** (bool) –
- **drop\_edge\_attrs** (bool) –
- **verbose** (bool) –
- **direct** (bool) –
- **engine** (str) –

- **npartitions** (Optional[int]) –
- **chunksize** (Optional[int]) –

graphistry.pygraphistry.**infer\_labels** (*self*)

**Returns** Plotter w/neo4j

- Prefers point\_title/point\_label if available
- Fallback to node id
- Raises exception if no nodes available, no likely candidates, and no matching node id fallback

### Example

```
import graphistry
g = graphistry.nodes(pd.read_csv('nodes.csv'), 'id_col').infer_labels()
g.plot()
```

graphistry.pygraphistry.**layout\_settings** (*play=None, locked\_x=None, locked\_y=None, locked\_r=None, left=None, top=None, right=None, bottom=None, lin\_log=None, strong\_gravity=None, dissuade\_hubs=None, edge\_influence=None, precision\_vs\_speed=None, gravity=None, scaling\_ratio=None*)

Set layout options. Additive over previous settings.

Corresponds to options at <https://hub.graphistry.com/docs/api/1/rest/url/#urloptions>

### Example: Animated radial layout

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a', 'b', 'c', 'd'], 'boss': ['c', 'c', 'e', 'e']})
nodes = pd.DataFrame({
    'n': ['a', 'b', 'c', 'd', 'e'],
    'y': [1, 1, 2, 3, 4],
    'x': [1, 1, 0, 0, 0],
})
g = (graphistry
     .edges(edges, 's', 'd')
     .nodes(nodes, 'n')
     .layout_settings(locked_r=True, play=2000)
     .plot())
```

### Parameters

- **play** (Optional[int]) –
- **locked\_x** (Optional[bool]) –
- **locked\_y** (Optional[bool]) –
- **locked\_r** (Optional[bool]) –
- **left** (Optional[float]) –
- **top** (Optional[float]) –
- **right** (Optional[float]) –
- **bottom** (Optional[float]) –

- **lin\_log** (Optional[bool]) –
- **strong\_gravity** (Optional[bool]) –
- **dissuade\_hubs** (Optional[bool]) –
- **edge\_influence** (Optional[float]) –
- **precision\_vs\_speed** (Optional[float]) –
- **gravity** (Optional[float]) –
- **scaling\_ratio** (Optional[float]) –

`graphistry.pygraphistry.login(username, password, fail_silent=False)`

Authenticate and set token for reuse (api=3). If token\_refresh\_ms (default: 10min), auto-refreshes token. By default, must be reinvoked within 24hr.

`graphistry.pygraphistry.name(name)`

Upload name

**Parameters** `name` (*str*) – Upload name

`graphistry.pygraphistry.neptune(NEPTUNE_READER_HOST=None, NEPTUNE_READER_PORT=None, NEPTUNE_READER_PROTOCOL='wss', endpoint=None, gremlin_client=None)`

Provide credentials as arguments, as environment variables, or by providing a gremlinpython client  
Environment variable names are the same as the constructor argument names If endpoint provided, do not need host/port/protocol If no client provided, create (connect)

#### Example: Login and plot via parrams

```
import graphistry
(graphistry
 .neptune(
     NEPTUNE_READER_PROTOCOL='wss'
     NEPTUNE_READER_HOST='neptunedbcluster-xyz.cluster-ro-abc.us-east-
↪1.neptune.amazonaws.com'
     NEPTUNE_READER_PORT='8182'
 )
 .gremlin('g.E().sample(10)')
 .fetch_nodes() # Fetch properties for nodes
 .plot())
```

#### Example: Login and plot via env vars

```
import graphistry
(graphistry
 .neptune()
 .gremlin('g.E().sample(10)')
 .fetch_nodes() # Fetch properties for nodes
 .plot())
```

#### Example: Login and plot via endpoint

```
import graphistry
(graphistry
 .neptune(endpoint='wss://neptunedbcluster-xyz.cluster-ro-abc.us-east-
↪1.neptune.amazonaws.com:8182/gremlin')
```

(continues on next page)

(continued from previous page)

```
.gremlin('g.E().sample(10)')
.fetch_nodes()  # Fetch properties for nodes
.plot()
```

**Example: Login and plot via client**

```
import graphistry
(graphistry
 .neptune(gremlin_client=client)
 .gremlin('g.E().sample(10)')
 .fetch_nodes()  # Fetch properties for nodes
 .plot())
```

**Parameters**

- **NEPTUNE\_READER\_HOST** (Optional[str]) –
- **NEPTUNE\_READER\_PORT** (Optional[str]) –
- **NEPTUNE\_READER\_PROTOCOL** (Optional[str]) –
- **endpoint** (Optional[str]) –
- **gremlin\_client** (Optional[Any]) –

**Return type** *Plotter*`graphistry.pygraphistry.nodes(nodes, node=None, *args, **kwargs)`

Specify the set of nodes and associated data. If a callable, will be called with current Plotter and whatever positional+named arguments

Must include any nodes referenced in the edge list.

**Parameters** **nodes** (*Pandas dataframe or Callable*) – Nodes and their attributes.

**Returns** Plotter

**Return type** *Plotter*

**Example**

```
import graphistry

es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g = graphistry
  .bind(source='src', destination='dst')
  .edges(es)

vs = pandas.DataFrame({'v': [0,1,2], 'lbl': ['a', 'b', 'c']})
g = g.bind(node='v').nodes(vs)

g.plot()
```

**Example**

```
import graphistry

es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g = graphistry.edges(es, 'src', 'dst')
```

(continues on next page)



(continued from previous page)

```
vs = pandas.DataFrame({'v': [0,1,2], 'lbl': ['a', 'b', 'c']})
g = g.nodes(vs, 'v')

g.plot()
```

**Example**

```
:: import graphistry

def sample_nodes(g, n): return g._nodes.sample(n)

df = pandas.DataFrame({'id': [0,1,2], 'v': [1,2,0]})

graphistry .nodes(df, 'id') ..nodes(sample_nodes, n=2) ..nodes(sample_nodes, None, 2) # equivalent
.plot()
```

graphistry.pygraphistry.**nodexl** (*xls\_or\_url*, *source='default'*, *engine=None*, *verbose=False*)

**Parameters**

- **xls\_or\_url** – file/http path string to a nodexl-generated xls, or a pandas ExcelFile() object
- **source** – optionally activate binding by string name for a known nodexl data source ('twitter', 'wikimedia')
- **engine** – optionally set a pandas Excel engine
- **verbose** – optionally enable printing progress by overriding to True

graphistry.pygraphistry.**pipe** (*graph\_transform*, *\*args*, *\*\*kwargs*)

Create new Plotter derived from current

**Parameters** *graph\_transform* (*Callable*) –

**Example: Simple**

```
import graphistry

def fill_missing_bindings(g, source='src', destination='dst'):
    return g.bind(source=source, destination=destination)

graphistry
    .edges(pandas.DataFrame({'src': [0,1,2], 'd': [1,2,0]}))
    .pipe(fill_missing_bindings, destination='d') # binds 'src'
    .plot()
```

**Return type** Plottable

graphistry.pygraphistry.**protocol** (*value=None*)

Set or get the protocol ('http' or 'https'). Set automatically when using a server alias. Also set via environment variable GRAPHISTRY\_PROTOCOL.

graphistry.pygraphistry.**refresh** (*token=None*, *fail\_silent=False*)

Use self or provided JWT token to get a fresher one. If self token, internalize upon refresh.

```
graphistry.pygraphistry.register(key=None,          username=None,          password=None,
                                token=None,         server=None,          protocol=None,
                                api=None,           certificate_validation=None,
                                bolt=None,          token_refresh_ms=600000,
                                store_token_creds_in_memory=None,
                                client_protocol_hostname=None)
```

API key registration and server selection

Changing the key effects all derived Plotter instances.

Provide one of key (api=1,2) or username/password (api=3) or token (api=3).

#### Parameters

- **key** (*Optional[str]*) – API key (1.0 API).
- **username** (*Optional[str]*) – Account username (2.0 API).
- **password** (*Optional[str]*) – Account password (2.0 API).
- **token** (*Optional[str]*) – Valid Account JWT token (2.0). Provide token, or username/password, but not both.
- **server** (*Optional[str]*) – URL of the visualization server.
- **certificate\_validation** (*Optional[bool]*) – Override default-on check for valid TLS certificate by setting to True.
- **bolt** (*Union[dict, Any]*) – Neo4j bolt information. Optional driver or named constructor arguments for instantiating a new one.
- **protocol** (*Optional[str]*) – Protocol used to contact visualization server, defaults to “https”.
- **token\_refresh\_ms** (*int*) – Ignored for now; JWT token auto-refreshed on plot() calls.
- **store\_token\_creds\_in\_memory** (*Optional[bool]*) – Store username/password in-memory for JWT token refreshes (Token-originated have a hard limit, so always-on requires creds somewhere)
- **client\_protocol\_hostname** (*Optional[str]*) – Override protocol and host shown in browser. Defaults to protocol/server or envvar GRAPHISTRY\_CLIENT\_PROTOCOL\_HOSTNAME.

**Returns** None.

**Return type** None

#### Example: Standard (2.0 api by username/password)

```
import graphistry
graphistry.register(api=3, protocol='http', server='200.1.1.1', username=
↳ 'person', password='pwd')
```

#### Example: Standard (2.0 api by token)

```
import graphistry
graphistry.register(api=3, protocol='http', server='200.1.1.1', token='abc')
```

#### Example: Remote browser to Graphistry-provided notebook server (2.0)

```
import graphistry
graphistry.register(api=3, protocol='http', server='nginx', client_protocol_
↪hostname='https://my.site.com', token='abc')
```

**Example: Standard (1.0)**

```
import graphistry
graphistry.register(api=1, key="my api key")
```

`graphistry.pygraphistry.server` (*value=None*)

Get the hostname of the server or set the server using hostname or aliases. Also set via environment variable GRAPHISTRY\_HOSTNAME.

`graphistry.pygraphistry.settings` (*height=None, url\_params={}, render=None*)

`graphistry.pygraphistry.store_token_creds_in_memory` (*value=None*)

Cache credentials for JWT token access. Default off due to not being safe.

`graphistry.pygraphistry.style` (*bg=None, fg=None, logo=None, page=None*)

Creates a base plotter with some style settings.

For parameters, see `plotter.style`.

**Returns** Plotter

**Return type** *Plotter*

**Example**

```
import graphistry
graphistry.style(bg={'color': 'black'})
```

```
graphistry.pygraphistry.tigergraph(protocol='http', server='localhost', web_port=14240,
                                   api_port=9000, db=None, user='tigergraph',
                                   pwd='tigergraph', verbose=False)
```

Register Tigergraph connection setting defaults

**Parameters**

- **protocol** (*Optional[str]*) – Protocol used to contact the database.
- **server** (*Optional[str]*) – Domain of the database
- **web\_port** (*Optional[int]*) –
- **api\_port** (*Optional[int]*) –
- **db** (*Optional[str]*) – Name of the database
- **user** (*Optional[str]*) –
- **pwd** (*Optional[str]*) –
- **verbose** (*Optional[bool]*) – Whether to print operations

**Returns** Plotter

**Return type** *Plotter*

**Example: Standard**

```
import graphistry
tg = graphistry.tigergraph(protocol='https', server='acme.com', db='my_db',
↪user='alice', pwd='tigergraph2')
```

(continues on next page)

```
graphistry.pygraphistry.verify_token(token=None, fail_silent=False)
```

Return True iff current or provided token is still valid

Return type `bool`

## 1.4 graphistry.arrow\_uploader module

```
class graphistry.arrow_uploader.ArrowUploader(server_base_path='http://nginx',
                                              view_base_path='http://localhost',
                                              name=None,          description=None,
                                              edges=None,         nodes=None,
                                              node_encodings=None,
                                              edge_encodings=None, token=None,
                                              dataset_id=None,     metadata=None,
                                              certificate_validation=True)
```

Bases: `object`

`arrow_to_buffer` (*table*)

Parameters *table* (`Table`) –

property `certificate_validation`

`create_dataset` (*json*)

property `dataset_id`

Return type `str`

property `description`

Return type `str`

property `edge_encodings`

property `edges`

Return type `Table`

`g_to_edge_bindings` (*g*)

`g_to_edge_encodings` (*g*)

`g_to_node_bindings` (*g*)

`g_to_node_encodings` (*g*)

`login` (*username*, *password*)

`maybe_bindings` (*g*, *bindings*, *base*=`{}`)

property `metadata`

property `name`

Return type `str`

property `node_encodings`

property `nodes`

**Return type** Table

**post** (*as\_files=True, memoize=True*)

**as\_files deprecation plan:** Graphistry 2.34: Introduced Graphistry 2.35: Does nothing (runtime warning); all uploads are Files Graphistry 2.36: Remove flag

**Parameters**

- **as\_files** (bool) –
- **memoize** (bool) –

**post\_arrow** (*arr, graph\_type, opts=""*)

**Parameters**

- **arr** (Table) –
- **graph\_type** (str) –
- **opts** (str) –

**post\_arrow\_generic** (*sub\_path, tok, arr, opts=""*)

**Parameters**

- **sub\_path** (str) –
- **tok** (str) –
- **arr** (Table) –

**Return type** Response

**post\_edges\_arrow** (*arr=None, opts=""*)

**post\_edges\_file** (*file\_path, file\_type='csv'*)

**post\_file** (*file\_path, graph\_type='edges', file\_type='csv'*)

**post\_g** (*g, name=None, description=None*)

**post\_nodes\_arrow** (*arr=None, opts=""*)

**post\_nodes\_file** (*file\_path, file\_type='csv'*)

**refresh** (*token=None*)

**property server\_base\_path**

**Return type** str

**property token**

**Return type** str

**verify** (*token=None*)

**Return type** bool

**property view\_base\_path**

**Return type** str

## 1.5 graphistry.ArrowFileUploader module

**class** graphistry.ArrowFileUploader.**ArrowFileUploader** (*uploader*)

Bases: object

Implement file API with focus on Arrow support

Memoization in this class is based on reference equality, while plotter is based on hash. That means the plotter resolves different-identity value matches, so by the time ArrowFileUploader compares, identities are unified for faster reference-based checks.

**Example: Upload files with per-session memoization** `uploader : ArrowUploader arr : pa.Table afu = ArrowFileUploader(uploader)`

`file1_id = afu.create_and_post_file(arr)[0] file2_id = afu.create_and_post_file(arr)[0]`

`assert file1_id == file2_id # memoizes by default (memory-safe: weak refs)`

**Example: Explicitly create a file and upload data for it** `uploader : ArrowUploader arr : pa.Table afu = ArrowFileUploader(uploader)`

`file1_id = afu.create_file() afu.post_arrow(arr, file_id)`

`file2_id = afu.create_file() afu.post_arrow(arr, file_id)`

`assert file1_id != file2_id`

**create\_and\_post\_file** (*arr, file\_id=None, file\_opts={}, upload\_url\_opts='erase=true', memoize=True*)

Create file and upload data for it.

Default `upload_url_opts='erase=true'` throws exceptions on parse errors and deletes upload.

Default `memoize=True` skips uploading 'arr' when previously uploaded in current session

See File REST API for `file_opts` (file create) and `upload_url_opts` (file upload)

### Parameters

- **arr** (Table) –
- **file\_id** (Optional[str]) –
- **file\_opts** (dict) –
- **upload\_url\_opts** (str) –
- **memoize** (bool) –

**Return type** Tuple[str, dict]

**create\_file** (*file\_opts={}*)

Creates File and returns `file_id` str.

### Defaults:

- `file_type`: 'arrow'

See File REST API for `file_opts`

**Parameters** **file\_opts** (dict) –

**Return type** str

**post\_arrow** (*arr, file\_id, url\_opts='erase=true'*)

Upload new data to existing file id

Default `url_opts='erase=true'` throws exceptions on parse errors and deletes upload.

See File REST API for `url_opts` (file upload)

#### Parameters

- **arr** (Table) –
- **file\_id** (str) –
- **url\_opts** (str) –

Return type dict

**uploader:** Any = None

`graphistry.ArrowFileUploader.DF_TO_FILE_ID_CACHE: weakref.WeakKeyDictionary = <WeakKeyDict`

**NOTE:** Will switch to `pa.Table -> ...` when RAPIDS upgrades from pyarrow, which adds weakref support

**class** `graphistry.ArrowFileUploader.MemoizedFileUpload` (*file\_id, output*)  
Bases: object

#### Parameters

- **file\_id** (str) –
- **output** (dict) –

**file\_id:** str

**output:** dict

**class** `graphistry.ArrowFileUploader.WrappedTable` (*arr*)  
Bases: object

Parameters **arr** (Table) –

**arr:** `pyarrow.lib.Table`

`graphistry.ArrowFileUploader.cache_arr` (*arr*)

Hold reference to most recent memoization entries Hack until RAPIDS supports Arrow 2.0, when `pa.Table` becomes weakly referenceable





## 2.1 versioneer module



## INDICES AND TABLES

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



## A

`addStyle()` (*graphistry.pygraphistry.PyGraphistry static method*), 3  
`addStyle()` (*in module graphistry.pygraphistry*), 22  
`api_key()` (*graphistry.pygraphistry.PyGraphistry static method*), 4  
`api_token()` (*graphistry.pygraphistry.PyGraphistry static method*), 4  
`api_token()` (*in module graphistry.pygraphistry*), 22  
`api_token_refresh_ms()` (*graphistry.pygraphistry.PyGraphistry static method*), 4  
`api_version()` (*graphistry.pygraphistry.PyGraphistry static method*), 4  
`arr` (*graphistry.ArrowFileUploader.WrappedTable attribute*), 43  
`arrow_to_buffer()` (*graphistry.arrow\_uploader.ArrowUploader method*), 40  
`ArrowFileUploader` (*class in graphistry.ArrowFileUploader*), 42  
`ArrowUploader` (*class in graphistry.arrow\_uploader*), 40  
`authenticate()` (*graphistry.pygraphistry.PyGraphistry static method*), 4

## B

`bind()` (*graphistry.pygraphistry.PyGraphistry static method*), 4  
`bind()` (*in module graphistry.pygraphistry*), 22  
`bolt()` (*graphistry.pygraphistry.PyGraphistry static method*), 4  
`bolt()` (*in module graphistry.pygraphistry*), 22

## C

`cache_arr()` (*in graphistry.ArrowFileUploader*), 43  
`certificate_validation()` (*graphistry.arrow\_uploader.ArrowUploader property*), 40  
`certificate_validation()` (*graphistry.pygraphistry.PyGraphistry static method*), 5

`client_protocol_hostname()` (*graphistry.pygraphistry.PyGraphistry static method*), 5  
`client_protocol_hostname()` (*in module graphistry.pygraphistry*), 23  
`cosmos()` (*graphistry.pygraphistry.PyGraphistry static method*), 5  
`cosmos()` (*in module graphistry.pygraphistry*), 23  
`create_and_post_file()` (*graphistry.ArrowFileUploader.ArrowFileUploader method*), 42  
`create_dataset()` (*graphistry.arrow\_uploader.ArrowUploader method*), 40  
`create_file()` (*graphistry.ArrowFileUploader.ArrowFileUploader method*), 42  
`cypher()` (*graphistry.pygraphistry.PyGraphistry static method*), 5  
`cypher()` (*in module graphistry.pygraphistry*), 23

## D

`dataset_id()` (*graphistry.arrow\_uploader.ArrowUploader property*), 40  
`default()` (*graphistry.pygraphistry.NumpyJSONEncoder method*), 3  
`description()` (*graphistry.arrow\_uploader.ArrowUploader property*), 40  
`description()` (*graphistry.pygraphistry.PyGraphistry static method*), 6  
`description()` (*in module graphistry.pygraphistry*), 23  
`DF_TO_FILE_ID_CACHE` (*in module graphistry.ArrowFileUploader*), 43  
`drop_graph()` (*graphistry.pygraphistry.PyGraphistry static method*), 6  
`drop_graph()` (*in module graphistry.pygraphistry*), 24

## E

`edge_encodings()` (*graphistry.arrow\_uploader.ArrowUploader property*), 40

edges () (*graphistry.arrow\_uploader.ArrowUploader* property), 40

edges () (*graphistry.pygraphistry.PyGraphistry* static method), 6

edges () (*in module graphistry.pygraphistry*), 24

encode\_edge\_badge () (*graphistry.pygraphistry.PyGraphistry* static method), 6

encode\_edge\_badge () (*in module graphistry.pygraphistry*), 24

encode\_edge\_color () (*graphistry.pygraphistry.PyGraphistry* static method), 6

encode\_edge\_color () (*in module graphistry.pygraphistry*), 24

encode\_edge\_icon () (*graphistry.pygraphistry.PyGraphistry* static method), 7

encode\_edge\_icon () (*in module graphistry.pygraphistry*), 25

encode\_point\_badge () (*graphistry.pygraphistry.PyGraphistry* static method), 8

encode\_point\_badge () (*in module graphistry.pygraphistry*), 26

encode\_point\_color () (*graphistry.pygraphistry.PyGraphistry* static method), 8

encode\_point\_color () (*in module graphistry.pygraphistry*), 26

encode\_point\_icon () (*graphistry.pygraphistry.PyGraphistry* static method), 9

encode\_point\_icon () (*in module graphistry.pygraphistry*), 27

encode\_point\_size () (*graphistry.pygraphistry.PyGraphistry* static method), 10

encode\_point\_size () (*in module graphistry.pygraphistry*), 28

**F**

file\_id (*graphistry.ArrowFileUploader.MemoizedFileUpload* attribute), 43

**G**

g\_to\_edge\_bindings () (*graphistry.arrow\_uploader.ArrowUploader* method), 40

g\_to\_edge\_encodings () (*graphistry.arrow\_uploader.ArrowUploader* method), 40

g\_to\_node\_bindings () (*graphistry.arrow\_uploader.ArrowUploader* method), 40

g\_to\_node\_encodings () (*graphistry.arrow\_uploader.ArrowUploader* method), 40

graph () (*graphistry.pygraphistry.PyGraphistry* static method), 11

graph () (*in module graphistry.pygraphistry*), 29

graphistry.arrow\_uploader module, 40

graphistry.ArrowFileUploader module, 42

graphistry.plotter module, 3

graphistry.pygraphistry module, 3

gremlin () (*graphistry.pygraphistry.PyGraphistry* static method), 11

gremlin () (*in module graphistry.pygraphistry*), 29

gremlin\_client () (*graphistry.pygraphistry.PyGraphistry* static method), 11

gremlin\_client () (*in module graphistry.pygraphistry*), 29

gsq () (*graphistry.pygraphistry.PyGraphistry* static method), 11

gsq () (*in module graphistry.pygraphistry*), 29

gsq\_endpoint () (*graphistry.pygraphistry.PyGraphistry* static method), 13

gsq\_endpoint () (*in module graphistry.pygraphistry*), 31

**H**

hypergraph () (*graphistry.pygraphistry.PyGraphistry* static method), 13

hypergraph () (*in module graphistry.pygraphistry*), 31

**I**

infer\_labels () (*graphistry.pygraphistry.PyGraphistry* static method), 16

infer\_labels () (*in module graphistry.pygraphistry*), 34

layout\_settings () (*graphistry.pygraphistry.PyGraphistry* static method), 16

layout\_settings () (*in module graphistry.pygraphistry*), 34

login () (*graphistry.arrow\_uploader.ArrowUploader* method), 40

login () (*graphistry.pygraphistry.PyGraphistry* static method), 17

login () (*in module graphistry.pygraphistry*), 35

## M

maybe\_bindings() (*graphistry.arrow\_uploader.ArrowUploader* *method*), 40

MemoizedFileUpload (class in *graphistry.ArrowFileUploader*), 43

metadata() (*graphistry.arrow\_uploader.ArrowUploader* *property*), 40

module

- graphistry.arrow\_uploader*, 40
- graphistry.ArrowFileUploader*, 42
- graphistry.plotter*, 3
- graphistry.pygraphistry*, 3

## N

name() (*graphistry.arrow\_uploader.ArrowUploader* *property*), 40

name() (*graphistry.pygraphistry.PyGraphistry* *static method*), 17

name() (in module *graphistry.pygraphistry*), 35

neptune() (*graphistry.pygraphistry.PyGraphistry* *static method*), 17

neptune() (in module *graphistry.pygraphistry*), 35

node\_encodings() (*graphistry.arrow\_uploader.ArrowUploader* *property*), 40

nodes() (*graphistry.arrow\_uploader.ArrowUploader* *property*), 40

nodes() (*graphistry.pygraphistry.PyGraphistry* *static method*), 18

nodes() (in module *graphistry.pygraphistry*), 36

nodexl() (*graphistry.pygraphistry.PyGraphistry* *static method*), 19

nodexl() (in module *graphistry.pygraphistry*), 37

not\_implemented\_thunk()

- (*graphistry.pygraphistry.PyGraphistry* *static method*), 19

NumpyJSONEncoder (class in *graphistry.pygraphistry*), 3

## O

output (*graphistry.ArrowFileUploader.MemoizedFileUpload* *attribute*), 43

## P

pipe() (*graphistry.pygraphistry.PyGraphistry* *static method*), 19

pipe() (in module *graphistry.pygraphistry*), 37

Plotter (class in *graphistry.plotter*), 3

post() (*graphistry.arrow\_uploader.ArrowUploader* *method*), 41

post\_arrow() (*graphistry.arrow\_uploader.ArrowUploader* *method*), 41

post\_arrow() (*graphistry.ArrowFileUploader.ArrowFileUploader* *method*), 42

post\_arrow\_generic() (*graphistry.arrow\_uploader.ArrowUploader* *method*), 41

post\_edges\_arrow() (*graphistry.arrow\_uploader.ArrowUploader* *method*), 41

post\_edges\_file() (*graphistry.arrow\_uploader.ArrowUploader* *method*), 41

post\_file() (*graphistry.arrow\_uploader.ArrowUploader* *method*), 41

post\_g() (*graphistry.arrow\_uploader.ArrowUploader* *method*), 41

post\_nodes\_arrow() (*graphistry.arrow\_uploader.ArrowUploader* *method*), 41

post\_nodes\_file() (*graphistry.arrow\_uploader.ArrowUploader* *method*), 41

protocol() (*graphistry.pygraphistry.PyGraphistry* *static method*), 19

protocol() (in module *graphistry.pygraphistry*), 37

*PyGraphistry* (class in *graphistry.pygraphistry*), 3

## R

refresh() (*graphistry.arrow\_uploader.ArrowUploader* *method*), 41

refresh() (*graphistry.pygraphistry.PyGraphistry* *static method*), 19

refresh() (in module *graphistry.pygraphistry*), 37

register() (*graphistry.pygraphistry.PyGraphistry* *static method*), 20

register() (in module *graphistry.pygraphistry*), 37

relogin() (*graphistry.pygraphistry.PyGraphistry* *method*), 21

## S

server() (*graphistry.pygraphistry.PyGraphistry* *static method*), 21

server() (in module *graphistry.pygraphistry*), 39

server\_base\_path() (*graphistry.arrow\_uploader.ArrowUploader* *property*), 41

set\_bolt\_driver() (*graphistry.pygraphistry.PyGraphistry* *static method*), 21

settings() (*graphistry.pygraphistry.PyGraphistry* *static method*), 21

settings() (in module *graphistry.pygraphistry*), 39

store\_token\_creds\_in\_memory() (*graphistry.pygraphistry.PyGraphistry* *static method*), 21

store\_token\_creds\_in\_memory() (in module *graphistry.pygraphistry*), 39

`style()` (*graphistry.pygraphistry.PyGraphistry static method*), [21](#)

`style()` (*in module graphistry.pygraphistry*), [39](#)

## T

`tigergraph()` (*graphistry.pygraphistry.PyGraphistry static method*), [21](#)

`tigergraph()` (*in module graphistry.pygraphistry*), [39](#)

`token()` (*graphistry.arrow\_uploader.ArrowUploader property*), [41](#)

## U

`uploader` (*graphistry.ArrowFileUploader.ArrowFileUploader attribute*), [43](#)

## V

`verify()` (*graphistry.arrow\_uploader.ArrowUploader method*), [41](#)

`verify_token()` (*graphistry.pygraphistry.PyGraphistry static method*), [22](#)

`verify_token()` (*in module graphistry.pygraphistry*), [40](#)

`view_base_path()` (*graphistry.arrow\_uploader.ArrowUploader property*), [41](#)

## W

`WrappedTable` (*class in graphistry.ArrowFileUploader*), [43](#)