PyGraphistry Documentation

Release 0.11.5+0.gca71306.dirty

Graphistry, Inc.

Jun 24, 2020

Contents

Quickstart: Read our tutorial

Contents:

CHAPTER 1

graphistry package

1.1 Submodules

1.2 graphistry.plotter module

class graphistry.plotter.**Plotter**

Bases: object

Graph plotting class.

Created using `Graphistry.bind()`.

Chained calls successively add data and visual encodings, and end with a plot call.

To streamline reuse and replayable notebooks, Plotter manipulations are immutable. Each chained call returns a new instance that derives from the previous one. The old plotter or the new one can then be used to create different graphs.

The class supports convenience methods for mixing calls across Pandas, NetworkX, and IGraph.

bind(*source=None*, *destination=None*, *node=None*, *edge_title=None*, *edge_label=None*, *edge_color=None*, *edge_weight=None*, *edge_size=None*, *edge_opacity=None*, *edge_icon=None*, *edge_source_color=None*, *edge_destination_color=None*, *point_title=None*, *point_label=None*, *point_color=None*, *point_weight=None*, *point_size=None*, *point_opacity=None*, *point_icon=None*, *point_x=None*, *point_y=None*)

Relate data attributes to graph structure and visual representation.

To facilitate reuse and replayable notebooks, the binding call is chainable. Invocation does not effect the old binding: it instead returns a new Plotter instance with the new bindings added to the existing ones. Both the old and new bindings can then be used for different graphs.

Parameters

- **source** (*String.*) – Attribute containing an edge's source ID
- **destination** (*String.*) – Attribute containing an edge's destination ID

- **node** (*String.*) – Attribute containing a node's ID
- **edge_title** (*HtmlString.*) – Attribute overriding edge's minimized label text. By default, the edge source and destination is used.
- **edge_label** (*HtmlString.*) – Attribute overriding edge's expanded label text. By default, scrollable list of attribute/value mappings.
- **edge_color** (*int32 | int64.*) – Attribute overriding edge's color. rgba (int64) or int32 palette index, see palette definitions <https://graphistry.github.io/docs/legacy/api/0.9.2/api.html#extendedpalette>'_ for values. Based on Color Brewer.
- **edge_source_color** (*int64.*) – Attribute overriding edge's source color if no edge_color, as an rgba int64 value.
- **edge_destination_color** (*int64.*) – Attribute overriding edge's destination color if no edge_color, as an rgba int64 value.
- **edge_weight** (*String.*) – Attribute overriding edge weight. Default is 1. Advanced layout controls will relayout edges based on this value.
- **point_title** (*HtmlString.*) – Attribute overriding node's minimized label text. By default, the node ID is used.
- **point_label** (*HtmlString.*) – Attribute overriding node's expanded label text. By default, scrollable list of attribute/value mappings.
- **point_color** (*int32 | int64.*) – Attribute overriding node's color.rgba (int64) or int32 palette index, see palette definitions <https://graphistry.github.io/docs/legacy/api/0.9.2/api.html#extendedpalette>'_ for values. Based on Color Brewer.
- **point_size** (*HtmlString.*) – Attribute overriding node's size. By default, uses the node degree. The visualization will normalize point sizes and adjust dynamically using semantic zoom.
- **point_x** (*number.*) – Attribute overriding node's initial x position. Combine with ".settings(url_params={'play': 0}))" to create a custom layout
- **point_y** (*number.*) – Attribute overriding node's initial y position. Combine with ".settings(url_params={'play': 0}))" to create a custom layout

Returns Plotter.

Return type Plotter.

Example: Minimal

```
import graphistry
g = graphistry.bind()
g = g.bind(source='src', destination='dst')
```

Example: Node colors

```
import graphistry
g = graphistry.bind()
g = g.bind(source='src', destination='dst',
           node='id', point_color='color')
```

Example: Chaining

```
import graphistry
g = graphistry.bind(source='src', destination='dst', node='id')

g1 = g.bind(point_color='color1', point_size='size1')

g.bind(point_color='color1b')

g2a = g1.bind(point_color='color2a')
g2b = g1.bind(point_color='color2b', point_size='size2b')

g3a = g2a.bind(point_size='size3a')
g3b = g2b.bind(point_size='size3b')
```

In the above Chaining example, all bindings use src/dst/id. Colors and sizes bind to:

```
g: default/default
g1: color1/size1
g2a: color2a/size1
g2b: color2b/size2b
g3a: color2a/size3a
g3b: color2b/size3b
```

bolt (*driver*)

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

edges (*edges*)

Specify edge list data and associated edge attribute values.

Parameters **edges** – Edges and their attributes.

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()
```

graph (*ig*)

Specify the node and edge data.

Parameters **ig** (*NetworkX graph or an IGraph graph.*) – Graph with node and edge attributes.

Returns Plotter.

Return type Plotter.

gsql (*query*, *bindings={}*, *dry_run=False*)

Run Tigergraph query in interpreted mode and return transformed Plottable

param query Code to run

type query String.

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

type bindings Optional dictionary.

param dry_run Return target URL without running

type dry_run Bool, defaults to False

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

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()

gsql_endpoint(*method_name*, *args={}*, *bindings={}*, *db=None*, *dry_run=False*)

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

Parameters

- **method_name** (*String.*) – Stored procedure name
- **args** (*Optional dictionary.*) – Named endpoint arguments
- **bindings** (*Optional dictionary.*) – Mapping defining names of returned 'edges' and/or 'nodes', defaults to @@nodeList and @@edgeList
- **db** (*Optional string.*) – Name of the database, defaults to value set in .tigergraph(...)
- **dry_run** (*Bool, defaults to False*) – 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)
```

igraph2pandas(*ig*)

Under current bindings, transform an IGraph into a pandas edges dataframe and a nodes dataframe.

Example

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

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

ig = g.pandas2igraph(es)
ig.vs['community'] = ig.community_infomap().membership

(es2, vs2) = g.igraph2pandas(ig)
g.nodes(vs2).bind(point_color='community').plot()
```

networkx2pandas(*g*)

networkx_checkoverlap(*g*)

nodes(*nodes*)

Specify the set of nodes and associated data.

Must include any nodes referenced in the edge list.

Parameters **nodes** – 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()
```

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

pandas2igraph(*edges*, *directed=True*)

Convert a pandas edge dataframe to an IGraph graph.

Uses current bindings. Defaults to treating edges as directed.

Example

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

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

ig = g.pandas2igraph(es)
ig.vs['community'] = ig.community_infomap().membership
g.bind(point_color='community').plot(ig)
```

plot(*graph=None*, *nodes=None*, *name=None*, *render=None*, *skip_upload=False*)

Upload data to the Graphistry server and show as an iframe of it.

name, Uses the currently bound schema structure and visual encodings. Optional parameters override the current bindings.

When used in a notebook environment, will also show an iframe of the visualization.

Parameters

- **graph** (*Pandas dataframe, NetworkX graph, or IGraph graph.*) – Edge table or graph.
- **nodes** (*Pandas dataframe.*) – Nodes table.
- **render** (*Boolean*) – Whether to render the visualization using the native notebook environment (default True), or return the visualization URL

- **skip_upload** (*Boolean.*) – Return node/edge/bindings that would have been uploaded. By default, upload happens.

Example: Simple

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

Example: Shorthand

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

settings(*height=None*, *url_params={}*, *render=None*)

Specify iframe height and add URL parameter dictionary.

The library takes care of URI component encoding for the dictionary.

Parameters

- **height** (*Integer.*) – Height in pixels.
- **url_params** (*Dictionary*) – Dictionary of querystring parameters to append to the URL.
- **render** (*Boolean*) – Whether to render the visualization using the native notebook environment (default True), or return the visualization URL

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 string.*) – Protocol used to contact the database.
- **server** (*Optional string.*) – Domain of the database
- **web_port** (*Optional integer.*) –
- **api_port** (*Optional integer.*) –
- **db** (*Optional string.*) – Name of the database
- **user** (*Optional string.*) –
- **pwd** (*Optional string.*) –
- **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')
```

1.3 graphistry.pygraphistry module

Top-level import of class PyGraphistry as "Graphistry". Used to connect to the Graphistry server and then create a base plotter.

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 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_version(*value=None*)

Set or get the API version (1 or 2). 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.

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 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 edges(*edges*)

static graph(*ig*)

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 String.

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

type bindings Optional dictionary.

param dry_run Return target URL without running

type dry_run Bool, defaults to False

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

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-definend endpoint and return transformed Plottable

Parameters

- **method_name** (*String.*) – Stored procedure name
- **args** (*Optional dictionary.*) – Named endpoint arguments
- **bindings** (*Optional dictionary.*) – Mapping defining names of returned 'edges' and/or 'nodes', defaults to @@nodeList and @@edgeList
- **db** (*Optional string.*) – Name of the database, defaults to value set in .tigergraph(...)
- **dry_run** (*Bool, defaults to False*) – 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*)
Transform a dataframe into a hypergraph.

Parameters

- **raw_events** (*Dataframe*) – Dataframe to transform
- **entity_types** (*List*) – Optional list of 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

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

The transform creates a node for every row, and turns a row's column entries into node attributes. If direct=False (default), every unique value within a column is also turned into a node. Edges are added to connect a row's nodes to each of its column nodes, or if direct=True, to one another. Nodes are given the attribute 'type' corresponding to the originating column name, or in the case of a row, 'EventID'.

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.

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 Dictionary

Example

```
import graphistry
h = graphistry.hypergraph(my_df)
g = h['graph'].plot()
```

static login(*username*, *password*, *fail_silent=False*)

Authenticate and set token for reuse (api=3). Periodically call if risk of token going stale.

static nodes(*nodes*)

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 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 register(*key=None*, *username=None*, *password=None*, *token=None*, *server=None*, *protocol=None*, *api=None*, *certificate_validation=None*, *bolt=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** (*String.*) – API key.
- **server** (*Optional string.*) – URL of the visualization server.
- **protocol** (*Optional string.*) – Protocol used to contact visualization server

Returns None.

Return type None.

Example: Standard

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

Example: Developer

```
import graphistry
graphistry.register('my api key', server='staging', protocol='https')
```

Example: Through environment variable

:: export GRAPHISTRY_API_KEY = 'my api key'

:: import graphistry graphistry.register()

static server(*value=None*)
Get the hostname of the server or set the server using hostname or aliases. Supported aliases: 'localhost', 'staging', 'labs'. Also set via environment variable GRAPHISTRY_HOSTNAME.

static set_bolt_driver(*driver=None*)

static settings(*height=None*, *url_params={}*, *render=None*)

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 string.*) – Protocol used to contact the database.
- **server** (*Optional string.*) – Domain of the database
- **web_port** (*Optional integer.*) –
- **api_port** (*Optional integer.*) –
- **db** (*Optional string.*) – Name of the database
- **user** (*Optional string.*) –

- **pwd** (*Optional string.*) –
- **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')
```

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.**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.**edges**(*edges*)

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

graphistry.pygraphistry.**gsql**(*query*, *bindings=None*, *dry_run=False*)

Run Tigergraph query in interpreted mode and return transformed Plottable

param query Code to run

type query String.

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

type bindings Optional dictionary.

param dry_run Return target URL without running

type dry_run Bool, defaults to False

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

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-definend endpoint and return transformed Plottable

Parameters

- **method_name** (*String.*) – Stored procedure name
- **args** (*Optional dictionary.*) – Named endpoint arguments
- **bindings** (*Optional dictionary.*) – Mapping defining names of returned 'edges' and/or 'nodes', defaults to @@nodeList and @@edgeList
- **db** (*Optional string.*) – Name of the database, defaults to value set in .tigergraph(...)
- **dry_run** (*Bool, defaults to False*) – 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*)

Transform a dataframe into a hypergraph.

Parameters

- **raw_events** (*Dataframe*) – Dataframe to transform
- **entity_types** (*List*) – Optional list of 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

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

The transform creates a node for every row, and turns a row's column entries into node attributes. If direct=False (default), every unique value within a column is also turned into a node. Edges are added to connect a row's nodes to each of its column nodes, or if direct=True, to one another. Nodes are given the attribute 'type' corresponding to the originating column name, or in the case of a row, 'EventID'.

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.

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 Dictionary

Example

```
import graphistry
h = graphistry.hypergraph(my_df)
g = h['graph'].plot()
```

graphistry.pygraphistry.**nodes**(*nodes*)

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.**register**(*key=None*, *username=None*, *password=None*, *token=None*, *server=None*, *protocol=None*, *api=None*, *certificate_validation=None*, *bolt=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** (*String.*) – API key.
- **server** (*Optional string.*) – URL of the visualization server.
- **protocol** (*Optional string.*) – Protocol used to contact visualization server

Returns None.

Return type None.

Example: Standard

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

Example: Developer

```
import graphistry
graphistry.register('my api key', server='staging', protocol='https')
```

Example: Through environment variable

:: export GRAPHISTRY_API_KEY = 'my api key'

:: import graphistry graphistry.register()

graphistry.pygraphistry.**settings**(*height=None*, *url_params={}*, *render=None*)

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 string.*) – Protocol used to contact the database.
- **server** (*Optional string.*) – Domain of the database
- **web_port** (*Optional integer.*) –
- **api_port** (*Optional integer.*) –
- **db** (*Optional string.*) – Name of the database
- **user** (*Optional string.*) –
- **pwd** (*Optional string.*) –
- **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')
```

1.4 Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

P

R

S

T