brickschema Documentation

Gabe Fierro

Feb 21, 2024

CONTENTS

The `brickschema` package makes it easy to get started with Brick and Python. Among the features it provides are:

- management and querying of Brick models
- simple OWL-RL, SHACL and other inference
- **Haystack and VBIS integration:**
  - convert Haystack models to Brick
  - add VBIS tags to a Brick model, or get Brick types from VBIS tags

```python
import brickschema

# creates a new rdflib.Graph with a recent version of the Brick ontology
# preloaded.
g = brickschema.Graph(load_brick=True)
# OR use the absolute latest Brick:
# g = brickschema.Graph(load_brick_nightly=True)
# OR create from an existing model
# g = brickschema.Graph(load_brick=True).from_haystack(...)

# load in data files from your file system
g.load_file("mbuilding.ttl")
# ...or by URL (using rdflib)
g.parse("https://brickschema.org/ttl/soda_brick.ttl", format="ttl")

# perform reasoning on the graph (edits in-place)
g.expand(profile="shacl")

# validate your Brick graph against built-in shapes (or add your own)
valid, _, resultsText = g.validate()
if not valid:
    print("Graph is not valid!")
    print(resultsText)

# perform SPARQL queries on the graph
res = g.query("""SELECT ?afs ?afsp ?vav WHERE  {
    ?afs    a       brick:Air_Flow_Sensor .
    ?afsp   a       brick:Air_Flow_Setpoint .
    ?afs    brick:isPointOf ?vav .
    ?afsp   brick:isPointOf ?vav .
    ?vav    a   brick:VAV
}""")
for row in res:
    print(row)

# start a blocking web server with an interface for performing
# reasoning + querying functions
g.serve("localhost:8080")
# now visit in http://localhost:8080
```

CHAPTER

ONE

INSTALLATION

The `brickschema` package requires Python >= 3.7. It can be installed with `pip`:

```
pip install brickschema
```

CHAPTER

TWO

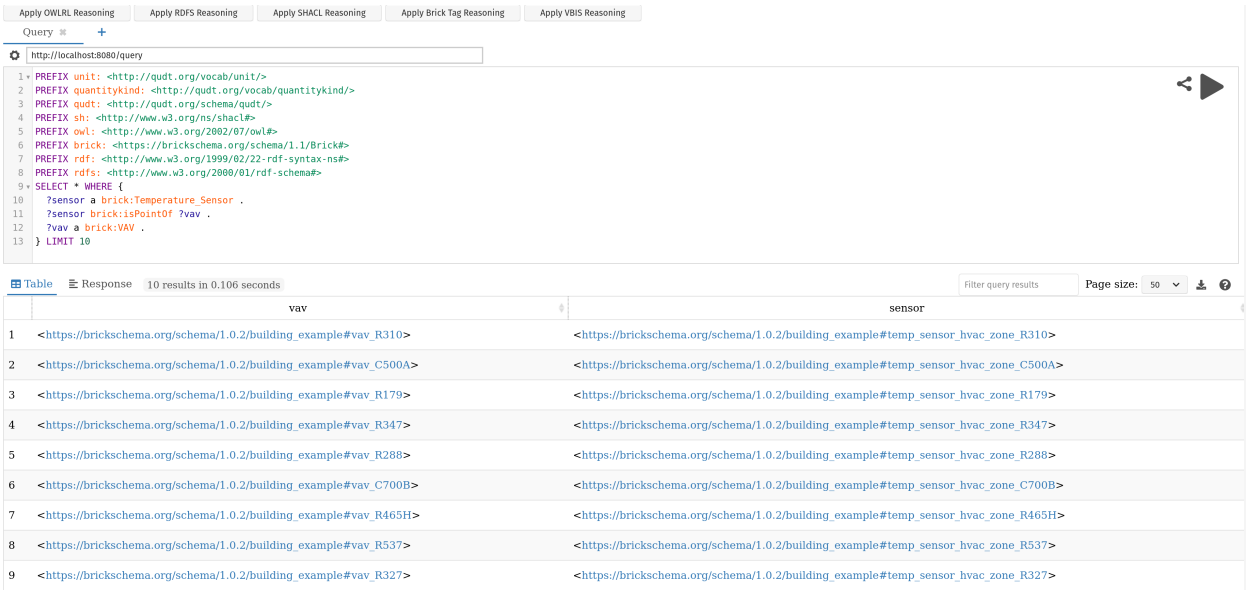
TABLE OF CONTENTS

2.1 Quick Feature Reference

2.1.1 Web Interface

brickschema incorporates a simple web server that makes it easy to apply inference and execute queries on Brick models. Call `.serve()` on a Graph object to start the webserver:

```
from brickschema import Graph
g = Graph(load_brick=True)
# load example Brick model
g.parse("https://brickschema.org/ttl/soda_brick.ttl")
g.serve("http://localhost:8080") # optional address argument
```



2.1.2 Brick Inference

Inference is the process of materializing all of the facts implied about a Brick model given the definitions in the Brick ontology. This process performs, among other things:

- **adding in "inverse" edges:**
  - Example: for all `brick:feeds`, add the corresponding `brick:isFedby`
- **annotating instances of classes with their Brick tags:**
  - Example: for all instances of `brick:Air_Temperature_Sensor`, add the mapped tags: `tag:Air`, `tag:Temperature`, `tag:Sensor` and `tag:Point`
- **annotating instances of classes with their measured substances and quantities:**
  - Example: for all instances of `brick:Air_Temperature_Sensor`, associate the `brick:Air` substance and `brick:Temperature` quantity
- **inferring which classes are implied by the available tags:**
  - Example: all entities with the `tag:Air`, `tag:Temperature`, `tag:Sensor` and `tag:Point` tags will be instantiated as members of the `brick:Air_Temperature_Sensor` class

The set of rules applied to the Brick model are defined formally here.

To apply the default inference process to your Brick model, use the `.expand()` method on the Graph.

```
from brickschema import Graph
bldg = Graph(load_brick=True)
bldg.load_file('mybuilding.ttl')
print(f"Before: {len(bldg)} triples")
bldg.expand("owlrl")
print(f"After: {len(bldg)} triples")
```

2.1.3 Haystack Inference

Requires a JSON export of a Haystack model First, export your Haystack model as JSON; we are using the public reference model *carytown.json*. Then you can use this package as follows:

```
import json
from brickschema import Graph
model = json.load(open("haystack-export.json"))
g = Graph(load_brick=True).from_haystack("http://project-haystack.org/carytown#", model)
points = g.query("""SELECT ?point ?type WHERE {
    ?point rdf:type/rdfs:subClassOf* brick:Point .
    ?point rdf:type ?type
}""")
print(points)
```

2.1.4 SQL ORM (experimental)

```
from brickschema.graph import Graph
from brickschema.namespaces import BRICK
from brickschema.orm import SQLORM, Location, Equipment, Point
# loads in default Brick ontology
g = Graph(load_brick=True)
# load in our model
g.load_file("test.ttl")
# put the ORM in a SQLite database file called "brick_test.db"
orm = SQLORM(g, connection_string="sqlite:///brick_test.db")
# get the points for each equipment
for equip in orm.session.query(Equipment):
    print(f"Equpiment {equip.name} is a {equip.type} with {len(equip.points)} points")
    for point in equip.points:
        print(f"    Point {point.name} has type {point.type}")
# filter for a given name or type
hvac_zones = orm.session.query(Location)\
                        .filter(Location.type==BRICK.HVAC_Zone)\
                        .all()
print(f"Model has {len(hvac_zones)} HVAC Zones")
```

2.2 Managing Metadata in Graphs

Graphs are the primary unit of management for Brick models. *brickschema* provides two ways of managing graphs and the triples inside them:

- as a single bag of triples (*Graph*, a subclass of *rdflib.Graph*)
- as a union of individually addressable bags of triples (*GraphCollection*, a subclass of *rdflib.Dataset*)

Both *Graph* and *GraphCollection* posess the ability to import triples from a variety of sources — online resources, local files, etc — and perform reasoning/inference, validation and querying over those triples. However, *Graph* does not provide any addressable subdivision of the ingested triples; once those triples are loaded into the graph, they are all considered part of the same flat set.

GraphCollection introduces a new method, *load_graph()*, which imports triples from the provided source into its *own graph*. This graph is an instance of *Graph* and can be queried, reasoned, validated just like other graphs. The name of the graph is a URI given by any *owl:Ontology* definition inside the graph (which can be overridden). The encapsulating *GraphCollection* object can query the constituent graphs individually, or as a union.

The advantage of *GraphCollection* over *Graph* is that it makes it easier to upgrade individual graphs — ontology definitions, building instances, etc — separately.

```
from brickschema import GraphCollection
from brickschema.namespaces import BRICK
from rdflib import Namespace

 # Create a new graph collection
 gc = GraphCollection(load_brick=True)
 # the Brick ontology is loaded under its own URI
 # the other graph is the "default" graph which contains
 # reasoned and inferred triples
```

(continues on next page)

(continued from previous page)

```
assert URIRef(BRICK) in g.graph_names
assert len(g.graph_names) == 2

brick = gc.graph(URIRef(BRICK))
# we can work with the Brick ontology graph independently
equipment_classes = brick.query("""
    SELECT ?equipment_class
    WHERE { ?equipment_class rdf:type brick:EquipmentClass }""")

# add a building graph with a sensor
EX1 = Namespace("urn:ex1#")
# referring to the graph implicitly creates it
bldg = gc.graph(EX1)
bldg.add((EX1["a"], A, BRICK["Sensor"]))

# perform SHACL reasoning on the graph; reasoned triples
# will be added to the default graph
gc.expand("shacl")

# now we can query the graph collection all together
assert len(g.graph_names) == 3
assert URIRef(EX1) in g.graph_names
assert URIRef(BRICK) in g.graph_names
res = gc.query("SELECT * WHERE { ?x a brick:Sensor }")
assert len(res) == 1, "Should have 1 sensor from adding graph"

# now we can replace the Brick ontology definition with a newer version
gc.remove_graph(URIRef(BRICK))
gc.load_graph("https://github.com/BrickSchema/Brick/releases/download/nightly/Brick.ttl
↪", graph_name=BRICK)
```

2.3 Second Example

```
from brickschema.graph import GraphCollection
from brickschema.namespaces import BRICK, A
from rdflib import URIRef, Namespace

# in-memory graph
g = GraphCollection()

# load Brick ontology
g.load_graph("https://sparql.gtf.fyi/ttl/Brick1.3rc1.ttl", format="turtle")

# declare namespace for the entities in the "instance" model
BLDG = Namespace("urn:building-instance/")

# grab the graph for the building instance model so we can add triples to it
bldg_graph = g.graph(URIRef(BLDG))

# now we can add triples to the building
```

(continues on next page)

(continued from previous page)

```
bldg_graph.add((BLDG["my-building"], A, BRICK.Building))
bldg_graph.add((BLDG["my-sensor"], A, BRICK.Zone_Air_Temperature_Sensor))

# when we run queries, run them against the "collection"
res = g.query("""SELECT * WHERE {
    ?sensor rdf:type/rdfs:subClassOf* brick:Temperature_Sensor
}""")
assert len(res) == 1

# we can save the building graph separately
bldg_graph = g.graph(URIRef(BLDG))
bldg_graph.serialize("my-building.ttl", format="turtle")
```

2.4 Persistent and Versioned Graphs

Tip: To use this feature, install brickschema with the "persistence" feature: *pip install brickschema[persistence]* or *pip install brickschema[all]*

The default *Graph* and *GraphCollection* classes implement in-memory stores that only track the latest version of the graph. Often, it is helpful to not just keep track of the history of how the graph has changed, but also to persist that graph so that it lasts beyond the lifetime of a program.

`PersistentGraph` is a subclass of *Graph* that implements a persistent store backed by RDFlib-SQLAlchemy. The contents of the graph are stored in a SQL database given by the connection string passed to the constructor:

```
from brickschema.persistent import PersistentGraph
from brickschema.namespaces import RDF, BRICK
# stores the graph in a local SQlite database
g = PersistentGraph("sqlite:///mygraph.sqlite", load_brick_nightly=True)
# PersistentGraph supports the full API of the normal transient Graph class
g.add((URIRef("http://example.org/mybuilding/ts1"), RDF.type, BRICK.Temperature_Sensor))
g.expand("shacl")
```

`VersionedGraphCollection` is another option which combines the persistence of *PersistentGraph*, the functionality of the base *Graph* class, and a transactional API for manipulating the graph. The versioned graph supports the following helpful features:

- undo/redo functionality for "rolling back" changes to the graph
- checkout a specific version of the graph at a point in time
- pre-commit and post-commit hooks for performing actions on the graph before and after the graph is committed

```
from brickschema.persistent import VersionedGraphCollection
from brickschema.namespaces import RDF, BRICK

g = VersionedGraphCollection("sqlite://") # in-memory

# can add precommit and postcommit hooks to implement desired functionality
# precommit hooks are run *before* the transaction is committed but *after* all of
# the changes have been made to the graph.
```

(continues on next page)

(continued from previous page)

```
# postcommit hooks are run *after* the transaction is committed.
def validate(graph):
    print("Validating graph")
    valid, _, report = graph.validate()
    assert valid, report
g.add_postcommit_hook(validate)

with g.new_changeset("my-building") as cs:
    # 'cs' is a rdflib.Graph that supports queries -- updates on it
    # are buffered in the transaction and cannot be queried until
    # the transaction is committed (at the end of the context block)
    cs.add((BLDG.vav1, A, BRICK.VAV))
    cs.add((BLDG.vav1, BRICK.feeds, BLDG.zone1))
    cs.add((BLDG.zone1, A, BRICK.HVAC_Zone))
    cs.add((BLDG.zone1, BRICK.hasPart, BLDG.room1))
print(f"Have {len(g)} triples")

snapshot = g.latest_version['timestamp']

with g.new_changeset("my-building") as cs:
    cs.remove((BLDG.zone1, A, BRICK.HVAC_Zone))
    cs.add((BLDG.zone1, A, BRICK.Temperature_Sensor))
print(f"Have {len(g)} triples")

# query the graph 3 seconds ago (before the latest commit)
print("Loop through versions")
for v in g.versions():
    print(f"{v.timestamp} {v.id} {v.graph}")
g = g.graph_at(timestamp=snapshot)
print(f"Have {len(g)} triples")
res = g.query("SELECT * WHERE { ?x a brick:Temperature_Sensor }")
num_results = len(list(res))
assert num_results == 0, num_results # should be 0 because sensor not added yet
```

Furthermore, the `VersionedGraphCollection` also acts like the *GraphCollection* where metadata can be managed.

```
from brickschema.persistent import VersionedGraphCollection
from brickschema.namespaces import BRICK, A
from rdflib import Namespace

vgc = VersionedGraphCollection(uri="sqlite://")

PROJECT = Namespace("https://example.org/my-project#")

# load Brick ontology
with vgc.new_changeset("Brick") as cs:
    cs.load_file("https://sparql.gtf.fyi/ttl/Brick1.3rc1.ttl")

# load other changes
with vgc.new_changeset("My-Project") as cs:
    cs.add((PROJECT["my-sensor"], A, BRICK.Zone_Air_Temperature_Sensor))
```

(continues on next page)

(continued from previous page)

```
res = vgc.query("""SELECT * WHERE {
    ?sensor rdf:type/rdfs:subClassOf* brick:Temperature_Sensor
}""")

# the query on the entire graph collection should find the sensor
assert len(res) == 1

g = vgc.graph_at(graph="My-Project")
res = g.query("""SELECT * WHERE {
    ?sensor rdf:type/rdfs:subClassOf* brick:Temperature_Sensor
}""")

# the same query but just on the graph "My-Project" should not have found any results
assert len(res) == 0

# serialize the graph without the Brick ontology
g.serialize("JustTheProject.ttl")
```

2.5 Inference

`brickschema` makes it easier to employ reasoning on your graphs. Simply call the `expand` method on the Graph object with one of the following profiles:

- `"rdfs"`: RDFS reasoning
- `"owlrl"`: OWL-RL reasoning (using 1 of 3 implementations below)
- `"vbis"`: add VBIS tags to Brick entities
- `"shacl"`: perform advanced SHACL reasoning

By default, `expand` will *simplify* the graph. Simplification is the process by which axiomatic, redundant or other "stray" triples are removed from the graph that may be added by a reasoner. This includes items like the following:

- triples that assert an entity to be an instance of `owl:Thing` or `owl:Nothing`
- triples that assert an entity to be a blank node
- triples that assert an entity to be the same as itself

To turn simplification off, simply add `simplify=False` when calling `expand`.

```
from brickschema import Graph

g = Graph(load_brick=True)
g.load_file("test.ttl")
g.expand(profile="owlrl")
print(f"Inferred graph has {len(g)} triples")
```

Brickschema also supports inference "schedules", where different inference regimes can be applied to a graph one after another. Specify a schedule by using + to join the profiles in the call to `expand`.

```
from brickschema import Graph

g = Graph(load_brick=True)
g.load_file("test.ttl")
# apply owlrl, shacl, vbis, then shacl again
g.expand(profile="owlrl+shacl+vbis+shacl")
print(f"Inferred graph has {len(g)} triples")
```

The package will automatically use the fastest available reasoning implementation for your system:

- `reasonable` (fastest, Linux-only for now): `pip install brickschema[reasonable]`
- `Allegro` (next-fastest, requires Docker): `pip install brickschema[allegro]`
- OWLRL (default, native Python implementation): `pip install brickschema`

To use a specific reasoner, specify `"reasonable"`, `"allegrograph"` or `"owlrl"` as the value for the `backend` argument to `graph.expand`.

2.6 Validate

The module utilizes the pySHACL package to validate a building ontology against the Brick Schema, its default constraints (shapes) and user provided shapes.

Please read **`Shapes Constraint Language (SHACL)`_** to see how it is used to validate RDF graphs against a set of constraints.

2.6.1 Example

```
from brickschema import Graph

g = Graph(load_brick=True)
g.load_file('myBuilding.ttl')
valid, _, report = g.validate()
print(f"Graph is valid? {valid}")
if not valid:
  print(report)

# validating using externally-defined shapes
external = Graph()
external.load_file("other_shapes.ttl")
valid, _, report = g.validate(shape_graphs=[external])
print(f"Graph is valid? {valid}")
if not valid:
  print(report)
```

2.6.2 Sample default shapes (in BrickShape.ttl)

```
# brick:hasLocation's object must be of brick:Location type
bsh:hasLocationRangeShape a sh:NodeShape ;
    sh:property [ sh:class brick:Location ;
        sh:message "Property hasLocation has object with incorrect type" ;
        sh:path brick:hasLocation ] ;
    sh:targetSubjectsOf brick:hasLocation .

# brick:isLocationOf's subject must be of brick:Location type
bsh:isLocationOfDomainShape a sh:NodeShape ;
    sh:class brick:Location ;
    sh:message "Property isLocationOf has subject with incorrect type" ;
    sh:targetSubjectsOf brick:isLocationOf .
```

2.7 Extensions and Alignments

The module makes it simple to list and load in extensions to the Brick schema, in addition to the alignments between Brick and other ontologies. These extensions are distributed as Turtle files on the Brick GitHub repository, but they are also pre-loaded into the *brickschema* module.

2.7.1 Listing and Loading Extensions

Extensions provide additional class definitions, rules and other augmentations to the Brick ontology.

```
from brickschema import Graph

g = Graph()
# returns a list of extensions
g.get_extensions()
# => ['shacl_tag_inference']

# loads the contents of the extension into the graph
g.load_extension('shacl_tag_inference')
# with this particular extension, you can now infer Brick
# classes from the tags associated with entities
g.expand("shacl")
```

2.7.2 Listing and Loading Alignments

Alignments define the nature of Brick's relationship to other RDF-based ontologies. For example, the Building Topology Ontology defines several location classes that are similar to Brick's; the alignment between BOT and Brick allows graphs defined in one language to be understood in the other.

Several Brick alignments are packaged with the *brickschema* module. These can be listed and dynamically loaded into a graph

```
from brickschema import Graph
```

(continues on next page)

(continued from previous page)

```
g = Graph()
# returns a list of alignments
g.get_alignments()
# => ['VBIS', 'REC', 'BOT']

# loads the contents of the alignment file into the graph
g.load_alignment('BOT')
# good idea to run a reasoner after loading in the extension
# so that the implied information is filled out
g.expand("owlrl")
```

2.8 Brick ORM

Tip: To use this feature, install brickschema with the "orm" feature: *pip install brickschema[orm]* or *pip install brickschema[all]*

Currently, the ORM models Locations, Points and Equipment and the basic relationships between them.

Please see the SQLAlchemy docs for detailed information on how to interact with the ORM. use the `orm.session` instance variable to interact with the ORM connection.

See querying docs for how to use the SQLalchemy querying mechanisms

2.8.1 Example

```
from brickschema.graph import Graph
from brickschema.namespaces import BRICK
from brickschema.orm import SQLORM, Location, Equipment, Point
# loads in default Brick ontology
g = Graph(load_brick=True)
# load in our model
g.load_file("test.ttl")
# put the ORM in a SQLite database file called "brick_test.db"
orm = SQLORM(g, connection_string="sqlite:///brick_test.db")
# get the points for each equipment
for equip in orm.session.query(Equipment):
    print(f"Equpiment {equip.name} is a {equip.type} with {len(equip.points)} points")
    for point in equip.points:
        print(f"    Point {point.name} has type {point.type}")
# filter for a given name or type
hvac_zones = orm.session.query(Location)\
                        .filter(Location.type==BRICK.HVAC_Zone)\
                        .all()
print(f"Model has {len(hvac_zones)} HVAC Zones")
```

2.9 brick_validate Command

The *brick_validate* command is similar to the pyshacl command with simplied command line arguments to validate a building ontology against the Brick Schema and Shapes Constraint Language (SHACL) contraints made for it.

When the validation results show contraint violations, the *brick_validate* command provides extra information associated with the violations in addition to the violation report by *pyshacl*. The extra infomation may be the *offending triple* or *violation hint*.

If no extra information is given for a reported violation, it means there is no appropriate handler for the perticular violation yet. If you think extra info is needed for the particular case, please open an issue with the brickschema module.

2.9.1 Example

```
# validate a building against the default shapes and extra shapes created by the uer
brick_validate myBuilding.ttl -s extraShapes.ttl
```

2.9.2 Sample output

```
Constraint violation:
[] a sh:ValidationResult ;
    sh:focusNode bldg:VAV2-3 ;
    sh:resultMessage "Must have at least 1 hasPoint property" ;
    sh:resultPath brick:hasPoint ;
    sh:resultSeverity sh:Violation ;
    sh:sourceConstraintComponent sh:MinCountConstraintComponent ;
    sh:sourceShape [ sh:message "Must have at least 1 hasPoint property" ;
         sh:minCount 1 ;
         sh:path brick:hasPoint ] .
 Violation hint (subject predicate cause):
 bldg:VAV2-3 brick:hasPoint "sh:minCount 1" .

 Constraint violation:
 [] a sh:ValidationResult ;
     sh:focusNode bldg:VAV2-4.DPR ;
     sh:resultMessage "Property hasPoint has object with incorrect type" ;
     sh:resultPath brick:hasPoint ;
     sh:resultSeverity sh:Violation ;
     sh:sourceConstraintComponent sh:ClassConstraintComponent ;
     sh:sourceShape [ sh:class brick:Point ;
          sh:message "Property hasPoint has object with incorrect type" ;
          sh:path brick:hasPoint ] ;
 sh:value bldg:Room-410 .
 Offending triple:
 bldg:VAV2-4.DPR brick:hasPoint bldg:Room-410 .
```

2.10 Merging Brick Models

Tip: To use this feature, install brickschema with the "merge" feature: *pip install brickschema[merge]* or *pip install brickschema[all]*

This module implements techniques for merging multiple Brick models into a single cohesive graph using techniques from a BuildSys 2020 paper. Eventually, other implementations may make their way into this module.

The module defines a *merge_type_cluster* function which interactively merges two Brick graphs together. This function finds instances in both graphs which are of the same type

2.10.1 Example

```
import brickschema
from brickschema.merge import merge_type_cluster
from rdflib import Namespace

# both graphs must have the same namespace
# AND must have RDFS.label for all entities
BLDG = Namespace("http://example.org/building/")

def validate(g):
    valid, _, report = g.validate()
    if not valid:
        raise Exception(report)

g1 = brickschema.Graph().load_file("bldg1")
#validate(g1)

g2 = brickschema.Graph().load_file("bldg2")
#validate(g2)

G = merge_type_cluster(g1, g2, BLDG, similarity_threshold=0.1)
validate(G)
G.serialize("merged.ttl", format="ttl")
```

2.11 Brickify tool

Tip: To use this feature, install brickschema with the "brickify" feature: *pip install brickschema[brickify]* or *pip install brickschema[all]*

The *brickify* tool is used to create Brick models from other data sources. It is installed as part of the `brickschema` package. If you installed py-brickschema from Github you may have usage examples included in the tests directory, otherwise, you can find them online in the test source tree.

The *brickify* tool is built around the notion of **handlers** and **operations**. **Handlers** are pieces of code (written in Python) that the *brickify* tool uses to carry out **operations** that transform data.

Handlers are how data is loaded by *brickify* and contain the code that executes the translations that are specified by the **operations**.

Operations are specified in a configuration file when the *brickify* tool is invoked by the user.

We expect that most users of *brickify* will not have to write a **Handler**, though they may need to write their own set of **operations**. Over time, we hope to include an expanded library of useful **Handlers** in *brickify* as well as example **operations** that can be easily customized for a particular job.

We expect a common scenario will be for *brickify* and the included **handlers** to be used as a tool in a building system integration job, where the **operations** might be written by the technical support team supporting the integration job, and then invoked by the field team against different data sources and building systems, with perhaps a small bit of customization.

2.11.1 Using Brickify

The *brickify* tool can be invoked on the command line as follows:

```
brickify sheet.tsv --output bldg.ttl --input-type tsv --config template.yml
```

where sheet.tsv might be a tabled stored in CSV/TSV file.

brickify starts with an empty graph, and uses **handlers** and **operations** to add the data from the input file (in this case, sheet.tsv) to the graph, and then write that graph out to a file (bldg.ttl)

For example, consider the following basic table with two rows that might be stored in `sheet.tsv`

VAV name	temperature sensor	temperature setpoint	has_reheat
A	A_ts	A_sp	false
B	B_ts	B_sp	true

Brickify selects the handler to use based on the input-type of the file. In this case, *brickify* will use the **TableHandler** to process the data.

Brickify loads the **operations** from the config file specified when *brickify* is run. The config file can be in either YAML or JSON, but for our examples we will use YAML. Here is an example *template.yml*

```
---
namespace_prefixes:
  brick: "https://brickschema.org/schema/Brick#"
operations:
  -
    data: |-
      bldg:{VAV name} rdf:type brick:VAV ;
                      brick:hasPoint bldg:{temperature sensor} ;
                      brick:hasPoint bldg:{temperature setpoint} .
      bldg:{temperature sensor} rdf:type brick:Temperature_Sensor .
      bldg:{temperature setpoint} rdf:type brick:Temperature_Setpoint .
  -
    conditions:
      - |
        '{has_reheat}'
    data: |-
      bldg:{VAV name} rdf:type brick:RVAV .
```

The above example configuration file has two **operations**. The first **operation** is a 'data' operation. In a 'data' operation, new data is added to the graph. In a dataset processed by a TableHandler, each operation is checked against each row of the input table. In a basic 'data' operation, if all of the variables mentioned in the operation are present in the row being processed, the body of the **operation** is updated using the values from the row being processed, and the data is inserted into the graph. The first **operation** above references the 'VAV_name', 'temperature sensor', and 'temperature setpoint' variables, and all of them are present in the first row, so the following data is inserted into the graph:

```
bldg:A rdf:type brick:VAV ;
                brick:hasPoint bldg:A_ts ;
                brick:hasPoint bldg:A_sp .
bldg:A_ts rdf:type brick:Temperature_Sensor .
bldg:A_sp rdf:type brick:Temperature_Setpoint .
```

Because the second row has all of the variables as well, the first operation is used again with the second row of the input file and the following information is inserted into the graph:

```
bldg:B rdf:type brick:VAV ;
                brick:hasPoint bldg:B_ts ;
                brick:hasPoint bldg:B_sp .
bldg:B_ts rdf:type brick:Temperature_Sensor .
bldg:B_sp rdf:type brick:Temperature_Setpoint .
```

The second **operation** in the file is a 'conditional' operation. A 'conditional' operation is much like a 'data' operation, and all of the variables specified in a 'conditional' operation must be present for the operation to be invoked, but a 'conditional' operation also includes an extra check to see if it should be used for a given row. In this case, the 'conditional' operation says that the has_reheat variable must be **true** in order for the associated 'data' operation to be invoked. In our example, the first row (for VAV 'A') under the column 'has_reheat' is listed as 'false' and so the 'data' operation does not fire. The second row (for VAV 'B') the 'has_reheat' column is 'true' and the 'data' operation fires, inserting the following triple into the graph

```
bldg:B a brick:RVAV .
```

The details of the 'conditional' syntax is detailed in the TableHandler section below.

2.11.2 Namespace and Prefix updates

Often, you would like to reuse a configuration file such as the 'template.yml' we used in our earlier examples, but you want to be able to customize them for a specific building or site. Brickify allows you to substitute a new namespace and RDF prefix for the building and site by using the command line. Brickify will replace the text from the template to be the new values on the command line.

```
brickify sheet.tsv --output bldg.ttl --input-type tsv --config template.yml --building-
↪prefix mybldg --building-namespace https://mysite.local/mybldg/#
```

Will produce in bldg.ttl:

```
@prefix brick: <https://brickschema.org/schema/Brick#> .
@prefix mybldg: <https://mysite.local/mybldg/#> .

mybldg:A a brick:VAV ;
    brick:hasPoint mybldg:A_sp,
        mybldg:A_ts .
```

2.11.3 Handler

The base Brickify Handler takes in an existing graph and updates it. The handler reads the entire graph into memory in one pass, and then runs each **operation** once against the entire graph. (Note that this is different than the TableHandler we were using in the example, which goes row-by-row through the input file, and runs the full set of operations against each row, e.g. if you have 3 rows and 2 operations, each of the 2 operations are run 3 times, once per row, for a total of 6 operations overall)

The base Handler is invoked when the `--input-format` option is set to `graph` or `rdf` or is left unspecified.

The supported **operations** for the base Handler are 'query' and 'data'. The 'query' operation executes a SPARQL update query to transform the input graph. Consider this example template.yml file:

```
---
namespace_prefixes:
  brick: "https://brickschema.org/schema/Brick#"
  yao: "https://example.com/YetAnotherOnology#"
operations:
  -
    query: |-
        DELETE {{
          ?vav a yao:vav .
        }}
        INSERT {{
          ?vav a brick:VAV .
        }}
        WHERE {{
          ?vav a yao:vav .
        }}
  -
    query: |-
        DELETE {{
          ?rvav a yao:vav_with_reheat .
        }}
        INSERT {{
          ?rvav a brick:RVAV .
        }}
        WHERE {{
          ?rvav a yao:vav_with_reheat .
        }}
```

This example has two **operations**, both of which are 'query' operations. Each operation is basically translating between one namespace and into another. The queries select a set of triples from the original graph, delete them from the original graph, and reinsert them into the new graph but in a new namespace.

2.11.4 Table Handler

The Table Handler processes input datasets row by row. The Table Halder is invoked with the `--input-format` is set to TSV, CSV, or table.

We have already seen parts of the TableHandler. Let's recall the config file we used earlier:

```
---
namespace_prefixes:
  brick: "https://brickschema.org/schema/Brick#"
operations:
  -
    data: |-
      bldg:{VAV name} rdf:type brick:VAV ;
                      brick:hasPoint bldg:{temperature sensor} ;
                      brick:hasPoint bldg:{temperature setpoint} .
      bldg:{temperature sensor} rdf:type brick:Temperature_Sensor .
      bldg:{temperature setpoint} rdf:type brick:Temperature_Setpoint .
  -
    conditions:
      - |
        '{has_reheat}'
    data: |-
      bldg:{VAV name} rdf:type brick:RVAV .
```

Internally, Brickify converts each 'data' operation to a SPARQL insert operation. If the 'data' operation fires, because all of the variables referenced in the operation are present in that row, Brickify executes a SPARQL *INSERT DATA* statement. This is the SPARQL generated from the first row:

```
INSERT DATA { bldg:A rdf:type brick:VAV ;
                brick:hasPoint bldg:A_ts ;
                brick:hasPoint bldg:A_sp .
bldg:A_ts rdf:type brick:Temperature_Sensor .
bldg:A_sp rdf:type brick:Temperature_Setpoint . }
```

Conditional syntax

Brickify implements conditions by taking the condition and feeding it to Python's `eval` method. If the condition evaluates to True, the data method fires, and if the method evaluates to False, the condition fails. Consider this input file:

VAV name	temperature sensor	temperature setpoint	has_reheat	thresh
A	A_ts	A_sp	false	16
B	B_ts	B_sp	true	12

One of the things that can be a little tricky with the 'condition' operation is ensuring that the types are correct when crossing from CSV/TSV and into Python, especially for strings and Booleans.

For example, this expression will fire for row A but not row B:

```
conditions:
  - |
    {thresh} > 14
```

Internally, this is converted to the string `'16 > 14'` and then passed to the Python `eval()` method, which returns `True`.

A trickier version - which looks like our earlier example but is *slightly* different:

```
conditions:
  - |
    {has_reheat}
```

In our example, this will fail! (Spoiler: we took away the quotes from our earlier example)

The issue is that the has_reheat column is pulled in as a string, but is not valid Python because the capitalization of 'true' and 'false' is incorrect in the TSV file.

One way to fix this is to correct the data:

VAV name	temperature sensor	temperature setpoint	has_reheat	thresh
A	A_ts	A_sp	False	16
B	B_ts	B_sp	True	12

This will match the condition because we have capitalized True and False. Unfortunately, changing the data in the input CSV you are processing may not always be possible.

As a compromise, to support this common use case where the input strings look like booleans but are not quite formatted right, Brickify expects Boolean conditions to be handled first as quoted strings:

```
conditions:
  - |
    '{has_reheat}'
```

Brickify will pass that code to the Python `eval()` method, which will return `'true'`, which is type `str` (and not `True` which is type `Boolean`) However, as a special case, Brickify converts the following strings to booleans: ["TRUE", "true", "True", "on", "ON"] all become `True`, and ["FALSE", "false", "False", "off", "OFF"] are converted to `False`.

An important note: the replacement text is not carried out on the substrings. At present, this will **not** work:

```
conditions:
  - |
    {thresh} > 12 and '{has_reheat}'
```

Template Operation

The TableHandler supports an additional operation, similar to the 'data' operation, that uses Jinja2 templates. This introduces a new section into the configuration file for defining Jinja2 templates, the 'macros' section, which is added at the top level of the configuration file.

The new **operation** is a 'template' operation, which can reference the Jinja2 macros from the top-level macro section. Much like 'data' operations, a 'template' operation only fires if all of the referenced variables are present in the row being processed.

Consider this input table:

VAV name	temperature sensor	temperature setpoint	has_reheat	sensors	setpoints
A	A_ts	A_sp	False	4	3
B	B_ts	B_sp	True	5	3

The example config file below defines two template operations. The template uses a 'for' loop to create multiple sensors and setpoints, following a naming pattern provided to macro as arguments. The numbers of sensors and setpoints come from the input CSV file.

```
---
namespace_prefixes:
  brick: "https://brickschema.org/schema/Brick#"
operations:
  -
    data: |-
      bldg:{VAV name}_0 rdf:type brick:VAV .
  -
    conditions:
      - |
        '{has_reheat}'
    data: |-
      bldg:{VAV name} rdf:type brick:RVAV .

  - template: |-
      {{ num_triples(value['VAV name'], "brick:hasPoint", value['temperature sensor'],␣
↪value['sensors'], "brick:Temperature_Sensor") }}

  - template: |-
      {{ num_triples(value['VAV name'], "brick:hasPoint", value['temperature setpoint'],␣
↪value['setpoints'], "brick:Temperature_Setpoint") }}

macros:
  - |-
    {% macro num_triples(subject, predicate, name, num, type) %}
        {% for i in range(num) %}
          bldg:{{ name }}_{{ i }} a {{ type }} .
          bldg:{{ subject }} {{ predicate }} bldg:{{ name }}_{{ i }} .
        {% endfor %}
    {% endmacro %}
```

And the output, just for the building B row:

```
bldg:B_ts_0 a brick:Temperature_Sensor .
bldg:B brick:hasPoint bldg:B_ts_0 .

bldg:B_ts_1 a brick:Temperature_Sensor .
bldg:B brick:hasPoint bldg:B_ts_1 .

bldg:B_ts_2 a brick:Temperature_Sensor .
bldg:B brick:hasPoint bldg:B_ts_2 .

bldg:B_ts_3 a brick:Temperature_Sensor .
bldg:B brick:hasPoint bldg:B_ts_3 .

bldg:B_ts_4 a brick:Temperature_Sensor .
bldg:B brick:hasPoint bldg:B_ts_4 .

bldg:B_sp_0 a brick:Temperature_Setpoint .
bldg:B brick:hasPoint bldg:B_sp_0 .
```

(continues on next page)

(continued from previous page)

```
bldg:B_sp_1 a brick:Temperature_Setpoint .
bldg:B brick:hasPoint bldg:B_sp_1 .

bldg:B_sp_2 a brick:Temperature_Setpoint .
bldg:B brick:hasPoint bldg:B_sp_2 .
```

2.11.5 Haystack Handler

The Haystack handler downloads Haystack files and converts them to Brick. It is invoked by using the `--input-type haystack` on the command line. The input file is a filepath or a URL where the Haystack TTL file can be found.

The conversion is carried out by the `HaystackRDFInferenceSession` method in this Python package.

2.12 brickschema package

2.12.1 Subpackages

2.12.2 Submodules

2.12.3 brickschema.bacnet module

brickschema.bacnet.**clean_name**(*name*)

brickschema.bacnet.**scan**(*ns: rdflib.namespace.Namespace = Namespace('urn:bacnet-scan/')*, *ip: Optional[str] = None*) → *brickschema.graph.Graph*

Scan for BACnet devices on the provided network. If no network is provided, use the default scan logic which scans all available interfaces. Provide a network by indicating the IP address that BAC0 should bind to

The Optional 'ns' parameter provides a namespace for the graph containing the scanned results.

2.12.4 brickschema.graph module

The *graph* module provides a wrapper class + convenience methods for building and querying a Brick graph

class brickschema.graph.**BrickBase**(*store: Union[Store, str] = 'default'*, *identifier: Optional[Union[_ContextIdentifierType, str]] = None*, *namespace_manager: Optional[NamespaceManager] = None*, *base: Optional[str] = None*, *bind_namespaces: _NamespaceSetString = 'rdflib'*)

Bases: `rdflib.graph.Graph`

expand(*profile*, *backend=None*, *simplify=True*, *ontology_graph=None*, *iterative=True*)

Expands the current graph with the inferred triples under the given entailment regime and with the given backend. Possible profiles are: - 'rdfs': runs RDFS rules - 'owlrl': runs full OWLRL reasoning - 'vbis': adds VBIS tags - 'shacl': does SHACL-AF reasoning (including tag inference, if the extension is loaded)

Possible backends are: - 'reasonable': default, fastest backend - 'allegrograph': uses Docker to interface with allegrograph - 'owlrl': native-Python implementation

Not all backend work with all profiles. In that case, brickschema will use the fastest appropriate backend in order to perform the requested inference.

To perform more than one kind of inference in sequence, use '+' to join the profiles:

import brickschema g = brickschema.Graph() g.expand(profile='rdfs+shacl') # performs RDFS inference, then SHACL-AF inference g.expand(profile='shacl+rdfs') # performs SHACL-AF inference, then RDFS inference

TODO: currently nothing is cached between expansions

get_alignments()

Returns a list of Brick alignments

This currently just lists the alignments already loaded into brickschema, but may in the future pull a list of alignments off of an online resolver

get_extensions()

Returns a list of Brick extensions

This currently just lists the extensions already loaded into brickschema, but may in the future pull a list of extensions off of an online resolver

get_most_specific_class(*classlist: List[rdflib.term.URIRef]*)

Given a list of classes (rdflib.URIRefs), return the 'most specific' classes This is a subset of the provided list, containing classes that are not subclasses of anything else in the list. Uses the class definitions in the graph to perform this task

Parameters **classlist** (*list of rdflib.URIRef*) – list of classes

Returns list of specific classes

Return type classlist (list of rdflib.URIRef)

rebuild_tag_lookup(*brick_file=None*)

Rebuilds the internal tag lookup dictionary used for Brick tag->class inference. This is broken out as its own method because it is potentially an expensive operation.

serve(*address='127.0.0.1:8080'*, *ignore_prefixes=[]*)

Start web server offering SPARQL queries and 1-click reasoning capabilities

Parameters

- **address** (*str*) – <host>:<port> of the web server
- **ignore_prefixes** (*list[str]*) – list of prefixes not to be added to the query editor's namespace bindings.

simplify()

Removes redundant and axiomatic triples and other detritus that is produced as a side effect of reasoning. Simplification consists of the following steps: - remove all "a owl:Thing", "a owl:Nothing" triples - remove all "a <blank node" triples - remove all "X owl:sameAs Y" triples

to_networkx()

Exports the graph as a NetworkX DiGraph. Edge labels are stored in the 'name' attribute :returns: networkx object representing this graph :rtype: graph (networkx.DiGraph)

validate(*shape_graphs=None*, *default_brick_shapes=True*, *engine: str = 'pyshacl'*)

Validates the graph using the shapes embedded w/n the graph. Optionally loads in normative Brick shapes and externally defined shapes

Parameters

- **shape_graphs** (*list of rdflib.Graph or* brickschema.graph.Graph) – merges these graphs and includes them in the validation
- **default_brick_shapes** (*bool*) – if True, loads in the default Brick shapes packaged with brickschema
- **engine** (*str*) – the SHACL engine to use. Options are 'pyshacl' and 'topquadrant'. Defaults to 'pyshacl'

Returns (conforms, resultsGraph, resultsText) from pyshacl

class brickschema.graph.**Graph**(**args*, *load_brick=False*, *load_brick_nightly=False*, *brick_version='1.3'*, *_delay_init=False*, ***kwargs*)

Bases: *brickschema.graph.BrickBase*

add(**triples*)

Adds triples to the graph. Triples should be 3-tuples of rdflib.Nodes (or alternatively 4-tuples if each triple has a context).

If the object of a triple is a list/tuple of length-2 lists/tuples, then this method will substitute a blank node as the object of the original triple, add the new triples, and add as many triples as length-2 items in the list with the blank node as the subject and the item[0] and item[1] as the predicate and object, respectively.

For example, calling add((X, Y, [(A,B), (C,D)])) produces the following triples:

```
X Y _b1 .
_b1 A B .
_b1 C D .
```

or, in turtle:

```
X Y [
  A B ;
  C D ;
] .
```

Otherwise, acts the same as rdflib.Graph.add

from_haystack(*namespace*, *model*)

Adds to the graph the Brick triples inferred from the given Haystack model. The model should be a Python dictionary produced from the Haystack JSON export

Parameters **model** (*dict*) – a Haystack model

from_triples(*triples*)

Creates a graph from the given list of triples

Parameters **triples** (*list of rdflib.Node*) – triples to add to the graph

load_alignment(*alignment_name*)

Loads the given alignment into the current graph by name. Use get_alignments() to get a list of alignments

load_extension(*extension_name*)

Loads the given extension into the current graph by name. Use get_extensions() to get a list of extensions

load_file(*filename=None*, *source=None*, *format=None*)

Imports the triples contained in the indicated file into the default graph.

Parameters

- **filename** (*str*) – relative or absolute path to the file

- **source** (*file*) – file-like object

property nodes

Returns all nodes in the graph

Returns nodes in the graph

Return type nodes (list of rdflib.URIRef)

class brickschema.graph.**GraphCollection**(**args*, *load_brick=False*, *load_brick_nightly=False*, *brick_version='1.3'*, ***kwargs*)

Bases: rdflib.graph.Dataset, *brickschema.graph.BrickBase*

contexts(*triple=None*)

Iterate over all contexts in the graph

If triple is specified, iterate over all contexts the triple is in.

property graph_names

Returns a list of the names of the graphs in the graph store

Returns list of graph names

Return type list

load_alignment(*alignment_name*)

Loads the given alignment into the current graph by name. Use get_alignments() to get a list of alignments

load_extension(*extension_name*)

Loads the given extension into the current graph by name. Use get_extensions() to get a list of extensions

load_graph(*filename: Optional[str] = None*, *source: Optional[io.IOBase] = None*, *format: Optional[str] = None*, *graph: Optional[rdflib.graph.Graph] = None*, *graph_name: Optional[rdflib.term.URIRef] = None*)

Imports the triples contained in the indicated file (or graph) into the graph. Names the graph using any owl:Ontology declaration found in the file or using the 'graph_name' argument if it is provided

Parameters

- **filename** (*str*) – relative or absolute path to the file
- **source** (*file*) – file-like object
- **graph** (*brickschema.Graph*) – graph to load into the collection
- **graph_name** (*rdflib.URIRef*) – name of the graph (defaults to owl:Ontology instance or 'default')

Returns the graph loaded from parsing the input

Return type parsed (rdflib.Graph)

remove_graph(*graph_name*)

Removes the named graph from the graph store

Parameters **graph_name** (*str*) – name of the graph to remove

2.12.5 brickschema.inference module

class brickschema.inference.**HaystackInferenceSession**(*namespace*)

Bases: *brickschema.inference.TagInferenceSession*

Wraps TagInferenceSession to provide inference of a Brick model from a Haystack model. The haystack model is expected to be encoded as a dictionary with the keys "cols" and "rows"; I believe this is a standard Haystack JSON export.

infer_entity(*tagset*, *identifier=None*, *equip_ref=None*)

Produces the Brick triples representing the given Haystack tag set

Parameters

- **tagset** (*list of str*) – a list of tags representing a Haystack entity
- **equip_ref** (*str*) – reference to an equipment if one exists

Keyword Arguments

- **identifier** (*str*) – if provided, use this identifier for the entity,
- **string.** (*otherwise, generate a random*) –

infer_model(*model*)

Produces the inferred Brick model from the given Haystack model :param model: a Haystack model :type model: dict

Returns

a Graph object containing the inferred triples in addition to the regular graph

Return type graph (*brickschema.graph.Graph*)

class brickschema.inference.**OWLRLAllegroInferenceSession**

Bases: object

Provides methods and an inferface for producing the deductive closure of a graph under OWL-RL semantics. WARNING this may take a long time

Uses the Allegrograph reasoning implementation

expand(*graph*)

Applies OWLRL reasoning from the Python owlrl library to the graph

Parameters **graph** (brickschema.graph.Graph) – a Graph object containing triples

class brickschema.inference.**OWLRLNaiveInferenceSession**

Bases: object

Provides methods and an inferface for producing the deductive closure of a graph under OWL-RL semantics. WARNING this may take a long time

expand(*graph*)

Applies OWLRL reasoning from the Python owlrl library to the graph

Parameters **graph** (brickschema.graph.Graph) – a Graph object containing triples

class brickschema.inference.**OWLRLReasonableInferenceSession**

Bases: object

Provides methods and an inferface for producing the deductive closure of a graph under OWL-RL semantics. WARNING this may take a long time

expand(*graph*)

Applies OWLRL reasoning from the Python reasonable library to the graph

Parameters **graph** (brickschema.graph.Graph) – a Graph object containing triples

class brickschema.inference.**TagInferenceSession**(*load_brick=True*, *brick_version='1.3'*, *rebuild_tag_lookup=False*, *approximate=False*, *brick_file=None*)

Bases: object

Provides methods and an interface for inferring Brick classes from sets of Brick tags. If you want to work with non-Brick tags, you will need to use a wrapper class (see HaystackInferenceSession)

expand(*graph*)

Infers the Brick class for entities with tags; tags are indicated by the *brick:hasTag* relationship. :param graph: a Graph object containing triples :type graph: brickschema.graph.Graph

lookup_tagset(*tagset*)

Returns the Brick classes and tagsets that are supersets OR subsets of the given tagsets

Parameters **tagset** (*list of str*) – a list of tags

most_likely_tagsets(*orig_s*, *num=- 1*)

Returns the list of likely classes for a given set of tags, as well as the list of tags that were 'leftover', i.e. not used in the inference of a class

Parameters

- **tagset** (*list of str*) – a list of tags
- **num** (*int*) – number of likely tagsets to be returned; -1 returns all

Returns a 2-element tuple containing (1) most_likely_classes (list of str): list of Brick classes and (2) leftover (set of str): list of tags not used

Return type results (tuple)

class brickschema.inference.**VBISTagInferenceSession**(*alignment_file=None*, *master_list_file=None*, *brick_version='1.3'*)

Bases: object

Add appropriate VBIS tag annotations to the entities inside the provided Brick model

Algorithm: - get all Equipment entities in the Brick model (VBIs currently only deals w/ equip)

Parameters

- **alignment_file** (*str*) – use the given Brick/VBIS alignment file. Defaults to a pre-packaged version.
- **master_list_file** (*str*) – use the given VBIS tag master list. Defaults to a pre-packaged version.
- **brick_version** (*string*) – the MAJOR.MINOR version of the Brick ontology to load into the graph. Only takes effect for the load_brick argument

Returns A VBISTagInferenceSession object

expand(*graph*)

Parameters **graph** (brickschema.graph.Graph) – a Graph object containing triples

lookup_brick_class(*vbistag*)

Returns all Brick classes that are appropriate for the given VBIS tag

Parameters **vbistag** (*str*) – the VBIS tag that we want to retrieve Brick classes for. Pattern search is not supported yet

Returns list of the Brick classes that match the VBIS tag

Return type brick_classes (list of rdflib.URIRef)

2.12.6 brickschema.merge module

The *merge* module implements data integration methods for merging Brick graphs together. This is based on techniques described in 'Shepherding Metadata Through the Building Lifecycle' published in BuildSys 2020

brickschema.merge.**cluster_by_type**(*g1*, *g2*, *namespace*)

brickschema.merge.**console_label**(*deduper: dedupe.api.ActiveMatching*) → None

Train a matcher instance (Dedupe, RecordLink, or Gazetteer) from the command line. Example

```
> deduper = dedupe.Dedupe(variables)
> deduper.prepare_training(data)
> dedupe.console_label(deduper)
```

brickschema.merge.**flatten_features**(*features*)

brickschema.merge.**get_common_types**(*g1*, *g2*, *namespace*)

Returns the list of types that are common to both graphs. A type is included if both graphs have instances of that type

brickschema.merge.**get_entity_feature_vectors**(*g*, *namespace*)

Returns a dictionary of features for each entity in graph 'g'.

Entities are any node with at least one *rdf:type* edge that is in the given namespace.

brickschema.merge.**merge_type_cluster**(*g1*, *g2*, *namespace*, *similarity_threshold=0.9*, *merge_types=None*)

brickschema.merge.**unify_entities**(*G*, *e1*, *e2*)

Replaces all instances of e2 with e1 in graph G

2.12.7 brickschema.namespaces module

The *namespaces* module provides pointers to standard Brick namespaces and related ontology namespaces wrapper class and convenience methods for a Brick graph

brickschema.namespaces.**bind_prefixes**(*graph*, *brick_version='1.3'*)

Associate common prefixes with the graph

2.12.8 brickschema.orm module

ORM for Brick

class brickschema.orm.**Equipment**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base

SQLAlchemy ORM class for BRICK.Equipment; see SQLORM class for usage

location

location_id

name

points

type

class brickschema.orm.**Location**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base

SQLAlchemy ORM class for BRICK.Location; see SQLORM class for usage

equipment

name

points

type

class brickschema.orm.**Point**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base

SQLAlchemy ORM class for BRICK.Point; see SQLORM class for usage

equipment

equipment_id

location

location_id

name

type

class brickschema.orm.**SQLORM**(*graph*, *connection_string='sqlite://brick_orm.db'*)

Bases: object

A SQLAlchemy-based ORM for Brick models.

Currently, the ORM models Locations, Points and Equipment and the basic relationships between them.

2.12.9 brickschema.persistent module

2.12.10 brickschema.tagmap module

brickschema.tagmap.**tagmap** = {'active': ['real'], 'ahu': ['AHU'], 'airhandlingequip': ['AHU'], 'airterminalunit': ['terminal', 'unit'], 'apparent': ['power'], 'atmospheric': ['pressure'], 'avg': ['average'], 'barometric': ['pressure'], 'chillermechanismtype': ['chiller'], 'cmd': ['command'], 'co': ['CO'], 'co2': ['CO2'], 'condenserlooptype': ['condenser'], 'cooling': ['cool'], 'coolingcoil': ['cool', 'coil', 'equip'], 'coolingonly': ['cool'], 'coolingtower': ['cool', 'tower', 'equip'], 'delta': ['differential'], 'device': ['equip'], 'economizing': ['economizer'], 'elec': ['electrical'], 'elecheat': ['heat'], 'equip': ['equipment'], 'evaporator': ['evaporative'], 'fcu': ['FCU'], 'freq': ['frequency'], 'fueloil': ['fuel', 'oil'], 'fueloilheating': ['heat'], 'fumehood': ['fume', 'hood'], 'heatexchanger': ['heat', 'exchanger', 'equip'], 'heating': ['heat'], 'heatingcoil': ['heat', 'coil', 'equip'], 'heatpump': ['heat', 'exchanger', 'equip'], 'heatwheel': ['heat', 'wheel'], 'hvac': ['HVAC'], 'lighting': ['lighting', 'equip'], 'lights': ['lighting'], 'lightsgroup': ['lighting'], 'luminous': ['luminance'], 'meterscopetype': ['meter', 'equip'], 'mixing': ['mixed'], 'naturalgas': ['natural', 'gas'], 'occ': ['occupied'], 'precipitation': ['rain'], 'roof': ['rooftop'], 'rooftop': ['rooftop'], 'rotaryscrew': ['compressor'], 'rtu': ['RTU'], 'sitemeter': ['meter', 'equip'], 'sp': ['setpoint'], 'state': ['status'], 'steamheating': ['heat'], 'submeter': ['meter', 'equip'], 'temp': ['temperature'], 'unocc': ['unoccupied'], 'variableairvolume': ['vav'], 'vav': ['VAV'], 'volt': ['voltage']}

get values for:

ahuZoneDeliveryType AHU airCooling Air airVolumeAdjustabilityType Air chilledBeam Chilled chilledBeamZone Chilled chilledWaterCooling Chilled chillerMechanismType Chiller condenserClosedLoop Condenser condenserCooling Condenser condenserLoopType Condenser condenserOpenLoop Condenser diverting Direction

2.12.11 brickschema.web module

Brickschema web module. This embeds a Flask webserver which provides a local web server with: - SPARQL interpreter + query result visualization - buttons to perform inference

TODO: - implement https://www.w3.org/TR/sparql11-protocol/ on /query

class brickschema.web.**Server**(*graph*, *ignore_prefixes=[]*)

Bases: object

apply_reasoning(*profile*)

bindings()

home()

query()

start(*address='localhost:8080'*)

2.12.12 Module contents

Python package *brickschema* provides a set of tools, utilities and interfaces for working with, developing and interacting with Brick models.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

I

L

M

N

O

P

Q

R

S

T

U

V