brickschema Documentation

Gabe Fierro

Jun 09, 2021

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

```
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="owlrl")
g.expand(profile="tag") # infers Brick classes from Brick tags

# 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 >= 3.6. It can be installed with `pip`:

```
pip install brickschema
```

CHAPTER TWO

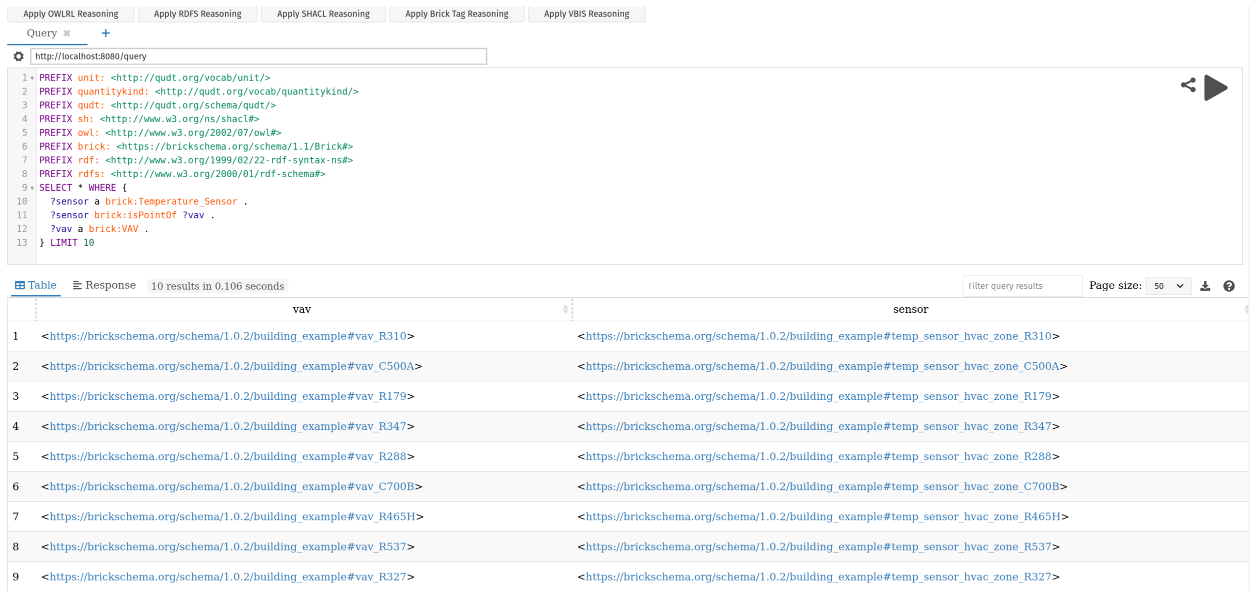
TABLE OF CONTENTS

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:

```python
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
```



```sparql
PREFIX unit: <http://qudt.org/vocab/unit/>
PREFIX quantitykind: <http://qudt.org/vocab/quantitykind/>
PREFIX qudt: <http://qudt.org/schema/qudt/>
PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX brick: <https://brickschema.org/schema/1.1/Brick#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT * WHERE {
  ?sensor a brick:Temperature_Sensor .
  ?sensor brick:isPointOf ?vav .
  ?vav a brick:VAV .
} LIMIT 10
```

	vav	sensor
1	<https://brickschema.org/schema/1.0.2/building_example#vav_R310>	<https://brickschema.org/schema/1.0.2/building_example#temp_sensor_hvac_zone_R310>
2	<https://brickschema.org/schema/1.0.2/building_example#vav_C500A>	<https://brickschema.org/schema/1.0.2/building_example#temp_sensor_hvac_zone_C500A>
3	<https://brickschema.org/schema/1.0.2/building_example#vav_R179>	<https://brickschema.org/schema/1.0.2/building_example#temp_sensor_hvac_zone_R179>
4	<https://brickschema.org/schema/1.0.2/building_example#vav_R347>	<https://brickschema.org/schema/1.0.2/building_example#temp_sensor_hvac_zone_R347>
5	<https://brickschema.org/schema/1.0.2/building_example#vav_R288>	<https://brickschema.org/schema/1.0.2/building_example#temp_sensor_hvac_zone_R288>
6	<https://brickschema.org/schema/1.0.2/building_example#vav_C700B>	<https://brickschema.org/schema/1.0.2/building_example#temp_sensor_hvac_zone_C700B>
7	<https://brickschema.org/schema/1.0.2/building_example#vav_R465H>	<https://brickschema.org/schema/1.0.2/building_example#temp_sensor_hvac_zone_R465H>
8	<https://brickschema.org/schema/1.0.2/building_example#vav_R537>	<https://brickschema.org/schema/1.0.2/building_example#temp_sensor_hvac_zone_R537>
9	<https://brickschema.org/schema/1.0.2/building_example#vav_R327>	<https://brickschema.org/schema/1.0.2/building_example#temp_sensor_hvac_zone_R327>

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

```python
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 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`.

```python
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`.

```python
from brickschema import Graph

g = Graph(load_brick=True)
g.load_file("test.ttl")
```

(continues on next page)

(continued from previous page)

```
# 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`.

2.3 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 Contraint Language (SHACL) to see how it is used to validate RDF graphs against a set of constraints.

2.3.1 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)
```

2.3.2 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
```

(continues on next page)

(continued from previous page)

```
bsh:isLocationOfDomainShape a sh:NodeShape ;
    sh:class brick:Location ;
    sh:message "Property isLocationOf has subject with incorrect type" ;
    sh:targetSubjectsOf brick:isLocationOf .
```

2.4 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.

2.4.1 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")
```

2.4.2 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

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")
```

2.5 Brick ORM

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

2.5.1 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")
```

2.6 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 Contraint 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.

2.6.1 Example

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

2.6.2 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 .
```

2.7 brickschema package

2.7.1 Subpackages

2.7.2 Submodules

2.7.3 brickschema.graph module

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

class brickschema.graph.**Graph**(**args, load_brick=False, load_brick_nightly=False, brick_version='1.2', **kwargs*)

Bases: `rdflib.graph.Graph`

add(**triples*)

Adds triples to the graph. Triples should be 3-tuples of rdflib.Nodes

If the last item 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 ;
>
>] .

`expand`(*profile=None*, *backend=None*, *simplify=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

`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

`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*)

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)

`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*)
Imports the triples contained in the indicated file into the 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)

`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'*)
Start web server offering SPARQL queries and 1-click reasoning capabilities

Parameters **`address`** (*`str`*) – <host>:<port> of the web server

`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

`validate`(*shape_graphs=None*, *default_brick_shapes=True*)
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

Returns (conforms, resultsGraph, resultsText) from pyshacl

2.7.4 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,
- **`generate a random string.`** (*`otherwise,`*) –

`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='1.2'*, *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='1.2'*)
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)

2.7.5 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='1.2'*`)`
    Associate common prefixes with the graph

2.7.6 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.

2.7.7 brickschema.tagmap module

`brickschema.tagmap.`**`tagmap = {'active': ['real'], 'ahu': ['AHU'], 'airhandlingequip': ['`**

# 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

2.7.8 brickschema.validate module

The *validate* module implements a wrapper of pySHACL to validate an ontology graph against the default Brick Schema constraints (called *shapes*) and user-defined shapes.

`class` `brickschema.validate.`**`Validator`**(*useBrickSchema=True*, *useDefaultShapes=True*, *brick_version='1.2'*)

Bases: `object`

Validates a data graph against Brick Schema and basic SHACL constraints for Brick. Allows extra constraints specific to the user's ontology.

`class Result`(*conforms*, *violationGraphs*, *textOutput*)

Bases: `object`

The type of returned object by validate() method

`validate`(*data_graph*, *shacl_graphs=[]*, *ont_graphs=[]*, *inference='rdfs'*, *abort_on_error=False*, *advanced=True*, *meta_shacl=True*, *debug=False*)

Validates data_graph against shacl_graph and ont_graph.

Parameters

- **`shacl_graphs`** – extra shape graphs in additon to BrickShape.ttl
- **`ont_graphs`** – extra ontology graphs in addtion to Brick.ttl

Returns object of Result class (conforms, violationGraphs, textOutput)

2.7.9 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*)

Bases: `object`

`apply_reasoning`(*profile*)

`home`()

`query`()

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

2.7.10 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

INDEX