Quepy Documentation

Release 0.1

Machinalis

January 16, 2017

Contents

```
  __ _ _    _  ____ _ __  _    _
 / _` | | | |/ _ \ '_ \| |  | |
| (_| | |_| |  __/ |_) | |_| |
 \__, |\__,_|\____| .__/ \__, |
    |_|            |_|    |___/
```

CHAPTER 1

What's quepy?

Quepy is a python framework to transform natural language questions to queries in a database query language. It can be easily customized to different kinds of questions in natural language and database queries. So, with little coding you can build your own system for natural language access to your database.

Currently **Quepy** provides support for Sparql and MQL query languages. We plan to extended it to other database query languages.

CHAPTER 2

Community

- Email us to `quepyproject@machinalis.com`
- Join our mailing list

CHAPTER 3

An example

To illustrate what can you do with quepy, we included an example application to access DBpedia contents via their *sparql* endpoint.

You can try the example online here: Online demo

Or, you can try the example yourself by doing:

```
python examples/dbpedia/main.py "Who is Tom Cruise?"
```

And it will output something like this:

```
SELECT DISTINCT ?x1 WHERE {
    ?x0 rdf:type foaf:Person.
    ?x0 rdfs:label "Tom Cruise"@en.
    ?x0 rdfs:comment ?x1.
}

Thomas Cruise Mapother IV, widely known as Tom Cruise, is an...
```

The transformation from natural language to sparql is done by first using a special form of regular expressions:

```
person_name = Group(Plus(Pos("NNP")), "person_name")
regex = Lemma("who") + Lemma("be") + person_name + Question(Pos("."))
```

And then using and a convenient way to express semantic relations:

```
person = IsPerson() + HasKeyword(person_name)
definition = DefinitionOf(person)
```

The rest of the transformation is handled automatically by the framework to finally produce this sparql:

```
SELECT DISTINCT ?x1 WHERE {
    ?x0 rdf:type foaf:Person.
    ?x0 rdfs:label "Tom Cruise"@en.
    ?x0 rdfs:comment ?x1.
}
```

Using a very similar procedure you could generate and MQL query for the same question obtaining:

```
[{
    "/common/topic/description": [{}],
    "/type/object/name": "Tom Cruise",
    "/type/object/type": "/people/person"
}]
```

CHAPTER 4

Installation

You need to have installed numpy. Other than that, you can just type:

```
pip install quepy
```

You can get more details on the installation here:

http://quepy.readthedocs.org/en/latest/installation.html

CHAPTER 5

Contents

5.1 Installation

5.1.1 Dependencies

- refo
- nltk - *if you intend to use nltk tagger*
- SPARQLWrapper *if you intend to use the examples*
- graphviz *if you intend to visualize your queries*

5.1.2 From pip

If you have **pip** installed you can run:

```
$ pip install quepy
```

then *Checking the installation*

5.1.3 From source code

Download the *GIT* repository from Github running:

```
$ git clone https://github.com/machinalis/quepy.git
```

run the install script doing:

```
$ cd quepy
$ sudo python setup.py install
```

and then *Checking the installation*

5.1.4 Checking the installation

To check if quepy was successfully installed run:

```
$ quepy --version
```

and you should obtain the version number.

5.1.5 Set up the POS tagger

After that you need to download the backend's POS tagger. It's ok if you don't know what that is, it's safe to treat it like a black box. Quepy uses nltk.

To set up quepy to be able to use nltk type:

```
$ quepy nltkdata /some/path/you/find/convenient
```

Also, every time you start a new app or use one, like the dbpedia example, you should configure *settings.py* to point to the path you chose.

5.2 Application Tutorial

Note: The aim of this tutorial is to show you how to build a custom natural language interface to your own database using an example.

To illustrate how to use quepy as a framework for natural language interface for databases, we will build (step by step) an example application to access DBpedia.

The finished example application can be tried online here: Online demo

The finished example code can be found here: Code

The first step is to select the questions that we want to be answered with dbpedia's database and then we will develop the quepy machinery to transform them into SPARQL queries.

5.2.1 Selected Questions

In our example application, we'll be seeking to answer questions like:

Who is <*someone*>, for example:

- Who is Tom Cruise?
- Who is President Obama?

What is <*something*>, for example:

- What is a car?
- What is the Python programming language?

List <*brand*> <*something*>, for example:

- List Microsoft software
- List Fiat cars

5.2.2 Starting a quepy project

To start a quepy project, you must create a quepy application. In our example, our application is called *dbpedia*, and we create the application by running:

```
$ quepy.py startapp dbpedia
```

You'll find out that a folder and some files where created. It should look like this:

```
$ cd dbpedia
$ tree .

.
-- dbpedia
|   -- __init__.py
|   -- parsing.py
|   -- dsl.py
|   -- settings.py
-- main.py

1 directory, 4 files
```

This is the basic structure of every quepy project.

- *dbpedia/parsing.py*: the file where you will define the regular expressions that will match natural language questions and transform them into an abstract semantic representation.
- *dbpedia/dsl.py*: the file where you will define the domain specific language of your database schema. In the case of SPARQL, here you will be specifing things that usually go in the ontology: relation names and such.
- *dbpedia/settings.py*: the configuration file for some aspects of the installation.
- *main.py*: this file is a optional kickstart point where you can have all the code you need to interact with your app. If you want, you can safely remove this file.

5.2.3 Configuring the application

First make sure you have already downloaded the necesary data for the nltk tagger. If not check the installation section.

Now edit *dbpedia/settings.py* and add the path to the nltk data to the *NLTK_DATA* variable. This file has some other configuration options, but we are not going to need them for this example.

Also configure the *LANGUAGE*, in this example we'll use `sparql`.

Note: What's a tagger anyway?

A "tagger" (in this context) is a linguistic tool help analyze natural language. It's composed of:

- A tokenizer
- A part-of-speech tagger
- A lemmatizer

If this is too much info for you, you can just treat it like a black box and it will be enough in the Quepy context.

5.2.4 Defining the regex

Note: To handle regular expressions, quepy uses refo, an awesome library to work with regular expressions as objects. You can read more about refo here.

We need to define the regular expressions that will match natural language questions and transform them into an abstract semantic representation. This will define specifically which questions the system will be able to handle and *what* to do with them.

In our example, we'll be editing the file *dbpedia/parsing.py*. Let's look at an example of regular expression to handle *"What is ..."* questions. The whole definition would look like this:

```
1  from refo import Group, Question
2  from quepy.dsl import HasKeyword
3  from quepy.parsing import Lemma, Pos, QuestionTemplate
4
5  from dsl import IsDefinedIn
6
7  class WhatIs(QuestionTemplate):
8      """
9      Regex for questions like "What is ..."
10     Ex: "What is a car"
11     """
12
13     target = Question(Pos("DT")) + Group(Pos("NN"), "target")
14     regex = Lemma("what") + Lemma("be") + target + Question(Pos("."))
15
16     def interpret(self, match):
17         thing = match.target.tokens
18         target = HasKeyword(thing)
19         definition = IsDefinedIn(target)
20         return definition
```

Now let's discuss this procedure step by step.

First of all, note that regex handlers need to be a subclass from *quepy.parsing.QuestionTemplate*. They also need to define a class attribute called `regex` with a refo regex.

Then, we describe the structure of the input question as a regular expression, and store it in the *regex* attribute. In our example, this is done in Line 14:

```
regex = Lemma("what") + Lemma("be") + target + Question(Pos("."))
```

This regular expression matches questions of the form "what is X?", but also "what was X?", "what were X?" and other variants of the verb to be because it is using the *lemma* of the verb in the regular expression. Note that the X in the question is defined by a variable called *target*, that is defined in Line 13:

```
target = Question(Pos("DT")) + Group(Pos("NN"), "target")
```

The *target* variable matches a string that will be passed on to the semantics to make part of the final query. In this example, we define that we want to match optionally a determiner (DT) followed by a noun (NN) labeled as "target".

Note that quepy can access different levels of linguistic information associated to the words in a question, namely their lemma and part of speech tag. This information needs to be associated to questions by analyzing them with a tagger.

Finally, if a regex has a successful match with an input question, the `interpret` method will be called with the match. In Lines 16 to 22, we define the *interpret* method, which specifies the semantics of a matched question:

```
def interpret(self, match):
    thing = match.target.tokens
    target = HasKeyword(thing)
    definition = IsDefinedIn(target)
    return definition
```

In this example, the contents of the target variable are the argument of a *HasKeyword* predicate. The *HasKeyword* predicate is part of the vocabulary of our specific database. In contrast, the *IsDefinedIn* predicate is part of the abstract semantics component that is described in the next section.

5.2.5 Defining the domain specific language

Quepy uses an abstract semantics as a language-independent representation that is then mapped to a query language. This allows your questions to be mapped to different query languages in a transparent manner.

In our example, the domain specific language is defined in the file *dbpedia/dsl.py*.

Let's see an example of the dsl definition. The predicate *IsDefinedIn* was used in line 21 of the previous example:

```
definition = IsDefinedIn(target)
```

IsDefinedIn is defined in the *dsl.py* file as follows:

```
from quepy.dsl import FixedRelation

class IsDefinedIn(FixedRelation):
    relation = "rdfs:comment"
    reverse = True
```

This means that *IsDefinedIn* is a Relation where the subject has *rdf:comment*. By creating a quepy class, we provide a further level of abstraction on this feature which allows to integrate it in regular expressions seamlessly.

The `reverse` part of the deal it's not easy to explain, so bear with me. When we say `relation = "rdfs:comment"` and `definition = IsDefinedIn(target)` we are stating that we want

```
?target rdfs:comment ?definition
```

But how does the framework knows that we are not trying to say this?:

```
?definition rdfs:comment ?target
```

Well, that's where `reverse` kicks in. If you set it to `True` (it's `False` by default) you get the first situation, if not you get the second situation.

5.2.6 Using the application

With all that set, we can now use our application. In the *main.py* file of our example there are some lines of code to use the application.

```
import quepy
dbpedia = quepy.install("dbpedia")
target, query, metadata = dbpedia.get_query("what is a blowtorch?")
print query
```

This code should be enough to obtain the following query:

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX quepy: <http://www.machinalis.com/quepy#>

SELECT DISTINCT ?x1 WHERE {
  ?x0 quepy:Keyword "blowtorch".
  ?x0 rdfs:comment ?x1.
}
```

5.3 Library Reference

5.3.1 Main API

Implements the Quepy Application API

class quepy.quepyapp.**QuepyApp**(*parsing*, *settings*)
Provides the quepy application API.

get_queries(*question*)
Given *question* in natural language, it returns three things:

•the target of the query in string format

•the query

•metadata given by the regex programmer (defaults to None)

The queries returned corresponds to the regexes that match in weight order.

get_query(*question*)
Given *question* in natural language, it returns three things:

•the target of the query in string format

•the query

•metadata given by the regex programmer (defaults to None)

The query returned corresponds to the first regex that matches in weight order.

quepy.quepyapp.**install**(*app_name*)
Installs the application and gives an QuepyApp object

5.3.2 Domain Specific Language

Domain specific language definitions.

class quepy.dsl.**FixedDataRelation**(*data*)
Expression for a fixed relation. This is "A is related to Data" through the relation defined in *relation*.

class quepy.dsl.**FixedRelation**(*destination*, *reverse=None*)
Expression for a fixed relation. It states that "A is related to B" through the relation defined in *relation*.

class quepy.dsl.**FixedType**
Expression for a fixed type. This captures the idea of something having an specific type.

class quepy.dsl.**HasKeyword**(*data*)
Abstraction of an information retrieval key, something standarized used to look up things in the database.

5.3.3 Parsing

exception quepy.parsing.**BadSemantic**
Problem with the semantic.

class quepy.parsing.**Lemma**(*tag*)
Predicate to check if a word has an specific *lemma*.

quepy.parsing.**Lemmas**(*string*)
Returns a Predicate that catches strings with the lemmas mentioned on *string*.

class quepy.parsing.**Match**(*match*, *words*, *i=None*, *j=None*)
Holds the matching of the regex.

class quepy.parsing.**Pos**(*tag*)
Predicate to check if a word has an specific *POS* tag.

quepy.parsing.**Poss**(*string*)
Returns a Predicate that catches strings with the POS mentioned on *string*.

class quepy.parsing.**QuestionTemplate**
Subclass from this to implement a question handler.

interpret(*match*)
Returns the intermediate representation of the regex. *match* is of type *quepy.regex.Match* and is analogous to a python re match. It contains matched groups in the regular expression.

When implementing a regex one must fill this method.

class quepy.parsing.**Token**(*tag*)
Predicate to check if a word has an specific *token*.

quepy.parsing.**Tokens**(*string*)
Returns a Predicate that catches strings with the tokens mentioned on *string*.

class quepy.parsing.**WordList**(*words*)
A list of words with some utils for the user.

5.3.4 NLTK Tagger

Tagging using NLTK.

quepy.nltktagger.**run_nltktagger**(*string*, *nltk_data_path=None*)
Runs nltk tagger on *string* and returns a list of quepy.tagger.Word objects.

5.3.5 Expression

This file implements the Expression class.

Expression is the base class for all the semantic representations in quepy. It's meant to carry all the information necessary to build a database query in an abstract form.

By design it's aimed specifically to represent a SPARQL query, but it should be able to represent queries in other database languages too.

A (simple) SPARQL query can be thought as a subgraph that has to match into a larger graph (the database). Each node of the subgraph is a variable and every edge a relation. So in order to represent a query, Expression implements a this subgraph using adjacency lists.

Also, Expression instances are meant to be combined with each other somehow to make complex queries out of simple ones (this is one of the main objectives of quepy).

To do that, every Expression has a special node called the head, which is the target node (variable) of the represented query. All operations over Expression instances work on the head node, leaving the rest of the nodes intact.

So Expression graphs are not built by explicitly adding nodes and edges like any other normal graph. Instead they are built by a combination of the following basic operations:

- **__init__: When a Expression is instantiated a single solitary** node is created in the graph.

- **decapitate: Creates a blank node and makes it the new head of the** `Expression`. Then it adds an edge (a relation) linking this new head to the old one. So in a single operation a node and an edge are added. Used to represent stuff like `?x rdf:type ?y`.
- **add_data: Adds a relation into some constant data from the head**

  node of the `Expression`. Used to represent stuff like

  `?x rdf:label "John Von Neumann"`.
- **merge: Given two Expressions, it joins their graphs preserving**

  every node and every edge intact except for their `head` nodes. The `head` nodes are merged into a single node that is the new `head` and shares all the edges of the previous heads. This is used to combine two graphs like this:

```
A = ?x rdf:type ?y
B = ?x rdf:label "John Von Neumann"
```

  Into a new one:

```
A + B = ?x rdf:type ?y;
        ?x rdf:label "John Von Neumann"
```

You might be saying "Why?! oh gosh why you did it like this?!". The reasons are:

- It allows other parts of the code to build queries in a super intuive language, like `IsPerson() + HasKeyword("Russell")`. Go and see the DBpedia example.
- You can only build connected graphs (ie, no useless variables in query).
- You cannot have variable name clashes.
- You cannot build cycles into the graph (could be a con to some, a plus to other(it's a plus to me))
- There are just 3 really basic operations and their semantics are defined concisely without special cases (if you care for that kind of stuff (I do)).

class `quepy.expression.`**Expression**

add_data(*relation*, *value*)
Adds a `relation` to some constant `value` to the `head` of the Expression. `value` is recommended be of type: - `unicode` - `str` and can be decoded using the default encoding (settings.py) - A custom class that implements a `__unicode__` method. - It can *NEVER* be an `int`.

You should not use this to relate nodes in the graph, only to add data fields to a node. To relate nodes in a graph use a combination of merge and decapitate.

decapitate(*relation*, *reverse=False*)
Creates a new blank node and makes it the `head` of the Expression. Then it adds an edge (a `relation`) linking the the new head to the old one. So in a single operation a node and an edge are added. If `reverse` is `True` then the `relation` links the old head to the new head instead of the opposite (some relations are not commutative).

get_head()
Returns the index (the unique identifier) of the head node.

iter_edges(*node*)
Iterates over the pairs: `(relation, index)` which are the neighbors of `node` in the expression graph, where: - `node` is the index of the node (the unique identifier). - `relation` is the label of the edge between the nodes - `index` is the index of the neighbor (the unique identifier).

iter_nodes()
Iterates the indexes (the unique identifiers) of the Expression nodes.

merge(*other*)
Given other Expression, it joins their graphs preserving every node and every edge intact except for the `head` nodes. The `head` nodes are merged into a single node that is the new `head` and shares all the edges of the previous heads.

quepy.expression.**isnode**(*x*)

5.3.6 Generation

Code generation from an expression to a database language.

The currently supported languages are:

- MQL
- Sparql
- Dot: generation of graph images mainly for debugging.

quepy.generation.**get_code**(*expression*, *language*)
Given an expression and a supported language, it returns the query for that expression on that language.

5.3.7 MQL Generation

quepy.mql_generation.**choose_start_node**(*e*)
Choose a node of the *Expression* such that no property leading to a data has to be reversed (with !).

quepy.mql_generation.**generate_mql**(*e*)
Generates a MQL query for the *Expression e*.

quepy.mql_generation.**paths_from_root**(*graph*, *start*)
Generates paths from *start* to every other node in *graph* and puts it in the returned dictionary *paths*. ie.: *paths_from_node(graph, start)[node]* is a list of the edge names used to get to *node* form *start*.

quepy.mql_generation.**post_order_depth_first**(*graph*, *start*)
Iterate over the nodes of the graph (is a tree) in a way such that every node is preceded by it's childs. *graph* is a dict that represents the *Expression* graph. It's a tree too beacuse Expressions are trees. *start* is the node to use as the root of the tree.

quepy.mql_generation.**safely_to_unicode**(*x*)
Given an "edge" (a relation) or "a data" from an *Expression* graph transform it into a unicode string fitted for insertion into a MQL query.

quepy.mql_generation.**to_bidirected_graph**(*e*)
Rewrite the graph such that there are reversed edges for every forward edge. If an edge goes into a data, it should not be reversed.

5.3.8 Sparql Generation

Sparql generation code.

quepy.sparql_generation.**adapt**(*x*)

quepy.sparql_generation.**escape**(*string*)

quepy.sparql_generation.**expression_to_sparql**(*e*, *full=False*)

quepy.sparql_generation.**triple**(*a*, *p*, *b*, *indentation=0*)

5.3.9 Dot Generation

Dot generation code.

quepy.dot_generation.**adapt**(*x*)

quepy.dot_generation.**dot_arc**(*a*, *label*, *b*)

quepy.dot_generation.**dot_attribute**(*a*, *key*)

quepy.dot_generation.**dot_fixed_type**(*a*, *fixedtype*)

quepy.dot_generation.**dot_keyword**(*a*, *key*)

quepy.dot_generation.**dot_type**(*a*, *t*)

quepy.dot_generation.**escape**(*x*, *add_quotes=True*)

quepy.dot_generation.**expression_to_dot**(*e*)

5.4 Important Concepts

5.4.1 Part of Speech tagset

The POS tagset used by quepy it's the **Penn Tagset** as defined here.

5.4.2 Keywords

When doing queries to a database it's very common to have a unified way to obtain data from it. In quepy we called it keyword. To use the Keywords in a quepy project you must first configurate what's the relationship that you're using. You do this by defining the class attribute of the *quepy.dsl.HasKeyword*.

For example, if you want to use **rdfs:label** as Keyword relationship you do:

```
from quepy.dsl import HasKeyword
HasKeyword.relation = "rdfs:label"
```

If your Keyword uses language specification you can configure this by doing:

```
HasKeyword.language = "en"
```

Quepy provides some utils to work with Keywords, like `quepy.dsl.handle_keywords()`. This function will take some text and extract IRkeys from it. If you need to define some sanitize function to be applied to the extracted Keywords, you have define the *staticmethod* sanitize.

For example, if your IRkeys are always in lowercase, you can define:

```
HasKeyword.sanitize = staticmethod(lambda x: x.lower())
```

5.4.3 Particles

It's very common to find patterns that are repeated on several regex so quepy provides a mechanism to do this easily. For example, in the DBpedia example, a country it's used several times as regex and it has always the same interpretation. In order to do this in a clean way, one can define a Particle by doing:

```
class Country(Particle):
    regex = Plus(Pos("NN") | Pos("NNP"))

    def interpret(self, match):
        name = match.words.tokens.title()
        return IsCountry() + HasKeyword(name)
```

this 'particle' can be used to match thing in regex like this:

```
regex = Lemma("who") + Token("is") + Pos("DT") + Lemma("president") + \
    Pos("IN") + Country() + Question(Pos("."))
```

and can be used in the interpret() method just as an attribut of the match object:

```
def interpret(self, match):
    president = PresidentOf(match.country)
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

q

S

T

W